AFRL-RI-RS-TR-2008-137
Final Technical Report
May 2008

THE DESIGN OF A POLYMORPHOUS COGNITIVE AGENT ARCHITECTURE (PCAA)

Lockheed Martin Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S822

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-137 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/ /s/

CHRISTOPHER J. FLYNN
Work Unit Manager

JAMES A. COLLINS, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 2008	Final	Sep 04 - Oct 07

4. TITLE AND SUBTITLE

THE DESIGN OF A POLYMORPHOUS COGNITIVE AGENT ARCHITECTURE (PCAA)

5a. CONTRACT NUMBER
FA8750-04-C-0266

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62304E

6. AUTHOR(S)
Mohammed Amduka, Jon Russo, and Krishna Jha (Lockheed Martin Advanced Technology Laboratories); André DeHon (Unversity of Penn); Richard Lethin and Jonathan Springer (Reservoir Labs); Rajit Manohar (Cornell University); Rami Melhem (University of Pitt); Bob Wray, (Soar Technology); Christan Lebiere and Brad Best (CMU); Ted Belding (New Vectors)

5d. PROJECT NUMBER
S822

5e. TASK NUMBER
PC

5f. WORK UNIT NUMBER
AA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lockheed Martin Corporation, Advanced Technology Laboratories
3 Executive Campus, 6th Floor
Cherry Hill NJ 08002-0000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITB
525 Brooks Rd
Rome NY 13441-4505

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-137

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-3185

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Lockheed Martin Advanced Technology Laboratories (LM ATL) led a highly qualified and motivated team to research, develop, and validate the Polymorphous Cognitive Agent Architecture (PCAA) as part of DARPA's Architecture for Cognitive Information Processing (ACIP). Our team investigated essential "Cognitive Information Processing" aspects of the architecture ranging from pure cognitive research into the essentials of human level reasoning to computing infrastructures that are essential to be able to transition PCAA into application domains that have a crucial need for this technology.

PCAA is a dynamic, adaptive cognitive architecture that makes previously intractable approximation tasks tractable for NP-hard cognitive problems. PCAA consists of: linear composable cognitive agents, a cognitive mark-up language for cognitive behavior definition, a cognitive layer for derivation of cognitive services and specialized cognitive agents, and a next generation polymorphic hardware and software layer for runtime composition and instantiation of cognitive agents.

15. SUBJECT TERMS
Polymorphous, cognition, computing architecture

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Christopher J. Flynn
U	U	U	UU	218	19b. TELEPHONE NUMBER (*Include area code*) N/A

Abstract

Lockheed Martin Advanced Technology Laboratories (LM ATL) led a highly qualified and motivated team to research, develop, and validate the Polymorphous Cognitive Agent Architecture (PCAA) as part of DARPA's Architecture for Cognitive Information Processing (ACIP). Our team investigated essential "Cognitive Information Processing" aspects of the architecture ranging from pure cognitive research into the essentials of human level reasoning to computing infrastructures that are essential to be able to transition PCAA into application domains that have a crucial need for this technology.

PCAA is a dynamic, adaptive cognitive architecture that makes previously intractable approximation tasks tractable for NP-hard cognitive problems. PCAA consists of: linear composable cognitive agents, a cognitive mark-up language for cognitive behavior definition, a cognitive layer for derivation of cognitive services and specialized cognitive agents, and a next generation polymorphic hardware and software layer for runtime composition and instantiation of cognitive agents.

Our approach included a comprehensive concept study in the context of representative DoD challenge problems that have a clear and well-defined need for ACIP technology. PCAA application experiments demonstrated clear performance improvements over traditional computing architectures for cognitive processing for these applications. Our innovations include:

- Dynamically composable hardware and software with linear scalability for cognitive processing across a massively parallel hardware fabric for real time autonomous systems.
- A dynamically composed agent architecture that partitions reactive and predefined behaviors into linear lower level cognitive agents that tailor and adapt the overall behavior of the computing architecture to immediate mission needs.
- Run-time derived cognitive virtual machines to partition cognitive processing to a new generation of computing run-time configured hardware and software to allow for dynamic cognitive computing reconfiguration required to achieve reactive processing.

Our research was driven by DoD applications that have demonstrated needs for diverse cognitive processing that cannot be addressed by current computing hardware and software architectures. We demonstrated our end-to-end approach for two applications with direct DoD relevance: control of autonomous Unmanned Aerial Vehicles and Intelligence Analysis.

Table of Contents

List of Figures

List of Tables

Acknowledgement

The authors would like to thank DARPA, AFRL and government labs for fostering innovation in this needed area.

1. Executive Summary

We describe our design, the Polymorphic Cognitive Agent Architecture (PCAA), a holistic hardware-software architecture that satisfies the requirements of a cognitive information processing need of any system realized in a physical computing system.

Inspired by a model of human cognitive capacity, a major goal of PCAA is to design an architecture that integrates the result of two decades of research in cognitive science, psychology, linguistics, artificial intelligence algorithms and computer hardware architecture into a cohesive information processing system that will enable systems of the future to achieve unprecedented performance in real-time problem solving, reasoning and learning.

Cognitive architectures draw from our increasing body of experimental observations of human cognition and theories about intelligence. A cognitive architecture unifies these insights and theories and renders them in computer based models. These models can exhibit aspects of intelligent behavior such as reasoning, learning, and decision making. One objective of research into cognitive architectures is that such intelligence could exhibit generalized problem solving ability. While the quest for generalized problem solving ability is an early facet in the history of artificial intelligence, the accumulated body of results in this program is attracting a number of researchers facing the need for complex automated problem-solving in diverse fields.

This lofty goal requires the architecture to support, seamless processing on four orthogonal cognitive information processing dimensions.

- *Representation:* It must support problem and solution representational paradigms from symbolic through non-symbolic.
- *Concurrency and dynamism:* It must support multiple granularity of concurrency stretching from sequential through massively parallel.
- *Inference and learning:* It must support all inference and reasoning methodologies, from knowledge-based to perception-oriented.
- *Synthesis and prediction:* It must support forward problem-solving mechanisms and methodologies for synthesis or prediction from semantic data pyramid to iterative refinement to probabilistic and approximate computation.

Not surprisingly, designing such an architecture is a significant challenge. As such, the PCAA team approached the bigger, general architectural design problems by first solving smaller but diverse and hard kernel problems. We embarked multiple case studies focusing on much-simplified versions of Unmanned Aerial Vehicle (UAV) mission planning and evidence marshaling problems that are so critical and dear to approximating intelligence systems. To foster new ideas and unearth new challenges we performed the case studies on a loosely integrated architecture and prototype PCAA computing platform. The process of solving these diverse problems on the pilot PCAA architecture and computation platform provided architectural insights and pointers towards designing a full fledged cognitive information processing architecture in a general fashion. Early analysis results point toward substantial benefits from PCAA like architectures as a cognitive information processing architecture.

1.1 Cognitive Approach

The PCAA team proposed to provide such an architecture by seamlessly combining three cognitive information processing components (Figure 1):

- The Macro-cognition Component: providing knowledge-based reasoning over symbolically represented entities.
- The Proto Cognition Component: providing perception-inspired reasoning over sub-symbolically represented entities.
- The Micro Cognition Component: providing expertise-based reasoning over hybrid-representation entities, and bridge the Macro meta oriented approach with Proto mostly non-symbolic neural approach.

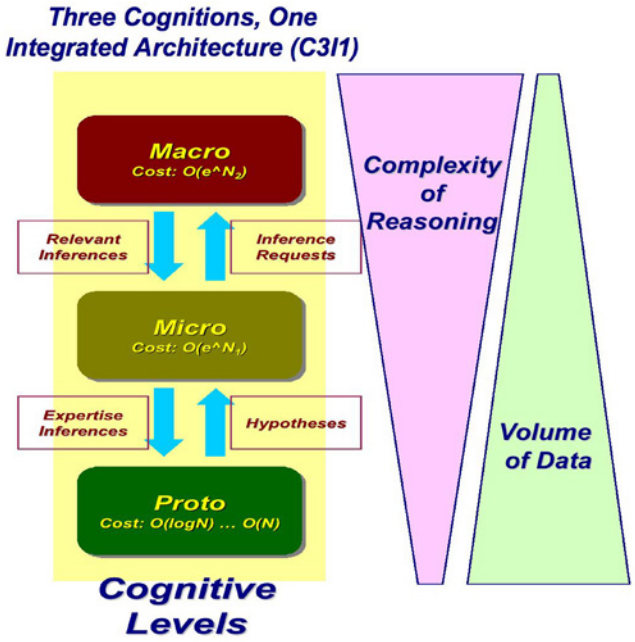


Figure 1. PCAA cognitive approach and the cognitive components

In addition, the PCAA team proposed to design and build innovative hardware to significantly speedup the basic cognitive information processing operations that can be factored out from general purpose algorithms in each of the components. These include:

- Direct hardware support for Programmable Objective Evaluation Memory, POEM
- Content addressable random access memory, CA-RAM
- Swarm processors: dynamically configurable massive parallelism for probabilistic, belief and message propagation. [PCAA-RecAP-TR]

1.2 Cognitive Framework

We have experimented with building solutions to small instances of hard problems, that are over and over again identified by the defense community as core applications, on a loosely integrated cognitive layer, coupled with a minimal implementation of the hardware layer. The results have

been encouraging. Further, we believe we showed these results can be significantly improved through tighter integration of the cognitive components and future smaller nano scale technology to support vast amounts of associative memories that are central to our architecture (Figure 2).

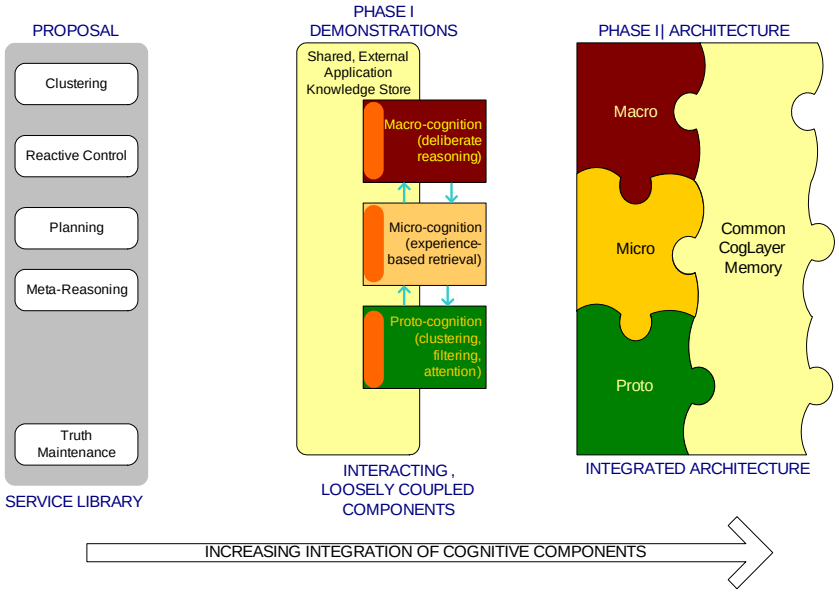


Figure 2. PCAA architectural development flow

The tighter integration of cognitive components entails a unified representation in memory. While a unified representation is a significant challenge that must not be underestimated, we believe near future hardware technologies will enable a unified representation. Such solution promises to significantly improve the seamless processing across multiple algorithmic dimensions, thereby significantly improving the architecture.

We have chosen to experiment with the SOAR [Newell91], ACT-R [Anderson98], and Swarming [Parunak02] models of cognition. SOAR is based on production systems in service of explicit goals; in each execution cycle, all productions whose precondition is enabled by the content of memory will fire, potentially updating the working memory or rule base. SOAR has designed-in mechanisms for resolving conflicts and integrating knowledge from multiple sources. ACT-R emphasizes reactive problem solving through expertise-based pattern matching. In our system, ACT-R also provides the interface between the macro-cognitive symbolic operations of SOAR, and the proto-cognitive sub-symbolic component implements perception processing and recognition, extracting information from massive amounts data.

Swarming systems are typically made out of simple agents, interacting locally without centralized control leading to an emergent behavior.

1.3 Discussion of Cognitive Approach

There is an apparent correspondence between our chosen cognitive frameworks and the case study application problems as framed algorithmically that arguably should provide problem

simplification. For example, swarming is a well-known meta-heuristic used in the solution of graph problems. The chunking operation in the Macro Cognition component of PCAA is an explanation based learning procedure that is analogous to the conflict clause generation procedure found in fast constraint solvers. The caching and approximate matching in micro cognition component of PCAA can leverage previous knowledge of good solutions that simplify the generation of new plans. But we were more ambitious in our desire to apply the cognitive frameworks; for example, we used Macro for reasoning about the problem in terms of an upper ontology for very high level problem simplification.

Our architecture is also an experiment into the fusion of the different cognitive models into one intelligence; we have prototyped a cognitive markup language (CML) and machine cognitive markup language (mCML) to implement communication between the intelligences. There are numerous opportunities for further investigation of such a configuration, such as finding ways to use the heuristics power of proto cognition or micro cognition to make the reasoning in Macro-cognition tractable. Furthermore, we extended the architecture to fuse other types of cognitive models outside of the PCAA realm, such as probabilistic reasoning to model uncertainty and the subsumption architectures.

Another challenge we faced was the degree and manner of integration of the existing algorithmic approaches in artificial intelligence (AI) to mission planning case study with the cognitive architectures we formed. It is an open question which is better: a fine-grained fusion of such techniques (e.g., implementing the search directly in one of our frameworks) vs. coarse partitioning (coupling a highly tuned constrained A* solver with a highly tuned implementation of a cognitive architecture). An implementation of A* in our cognitive frameworks for our case studies, for example, is attractive from the point of view of a seamless transition between that solution finding and the high level reasoning (justification), but it could be computationally very inefficient.

1.4 Hardware Support for the Micro Cognition Component

The Micro component employs memory intensive operation for expertise-based solution using three logical memory compartments: the declarative memory is used to symbolically represent compositional data, the procedural memory stores information for reaction in dynamic problem solving, and the working memory maintains the current operational context. Each memory type operates on similar principles for matching. The match operation is inexact in that the data associated with the closest key in memory (according to a definable distance metric) is retrieved. The similarity function, which compares an input operand to each content row (or so-called chunk) of memory may be further compounded by factors such as decay or practice.

Our hardware designs for the Micro component leveraged the inherent parallelism in the independent comparisons of chunks with input operands. Figure 3 shows the results of speedup vs. number of Power PCs cooperating on the match operation.

Further speedup (up to two orders of magnitude) was achieved by using a custom inexact matching architecture (implemented in a Xilinx Virtex-2 FPGA) shown in Figure 4.

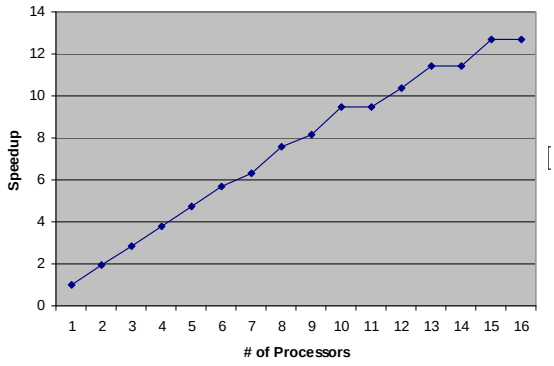


Figure 3. Speedup of ACT-R vs. Number of Processors

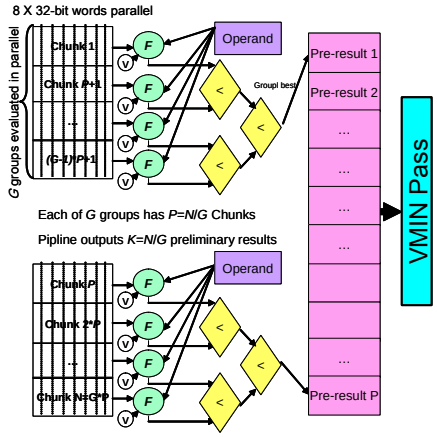


Figure 4. Matching Memory for ACT-R acceleration

The programmable match memory of Figure 4 implements a match function (F) in parallel across 256 bits of the input operand, along with parallel banking of memory, objective evaluation, and result selection. The implementation can sustain in excess of a 50 mega-chunks-per-second match rate for a fully pipelined match function.

1.5 Hardware Support for Macro Component

The core production matching of Macro targeted for hardware implementation is based on an algorithm known as Rete [Doorenbos95]. Rete is an exact-match algorithm that uses two memory types: the working memory contains facts about the world that are collected over time, and the production memory contains rules. Each rule in the production memory would typically be a set of conditions, and a set of actions to perform if those conditions are met. An added complexity is that the condition expressions may contain variables that must be bound during matching.

The study of Rete hardware targets a simulation of the event driven processing afforded by asynchronous logic [ManoharChandy04]. Figure 5 shows an asynchronous data flow network for a Rete, where each node is built from fine-grained logic, and can execute concurrently as soon as dependencies are met. As is typical with asynchronous approaches, energy consumption for computation is reduced by computing only when necessary, and performance can be greater by removing the need for a central clock.

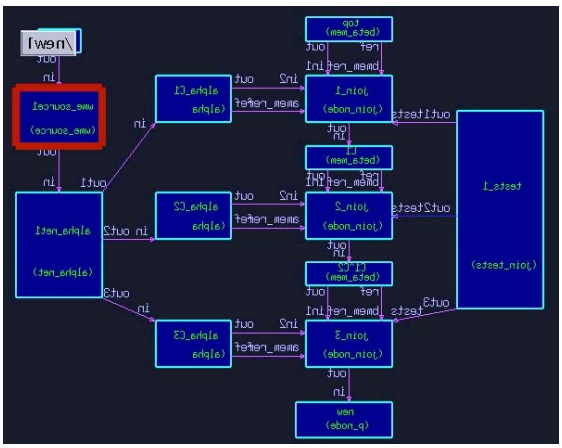


Figure 5. Rete in event-driven logic

Due to the dependencies of typical networks, the maximum concurrency achievable is only on the order of 10X to 20X. Hardware support for hash functions yields higher constant factors of acceleration and are under consideration.

1.6 Architectural Support for Proto Cognition

For most of the project's work, architectural support for Proto Cognition was limited to supporting swarming. We attempted to abstract the common control aspects of swarming algorithms through development of an API. Swarming is conceptually highly parallel. In our API prototyping, we found this parallelism to be readily available in practice. The challenge is often the grain size, however; with too little work per node, communication costs can dominate quickly. We have found it will be important to manage the grain size issue through clustering of highly-connected nodes and communication-aware layout of work, including dynamic rebalancing directly in hardware.

At the later stages of the project there was a realization that this layer is the interface between low level cognitive information processing, which we termed as perception or recognition and high level meta reasoning. We undertook a focused effort to define a low level hardware structure that efficiently supports this type of cognitive information processing known as perception. Our work is documented in a separate technical report [PCAA-RecAP-TR]. Here we summarize the findings and the conclusion.

We performed two case studies and identified two efficient architectures at hardware level for low level recognition and application level recognition. One or more meta level inference closes the perception loop. We performed experiments in the form of algorithm, dataset and application implementations in simulated and prototype hardware architectures consisting of off-the-shelf processors and FPGA boards. We compared these prototype hardware architecture performances with current state of the art architectures and methodologies and found the graph based runtime configurable architecture and its tight integration with active-inference memory based architecture to be an efficient hardware realization for proto cognitive component. In addition our results show these combinations of dynamically reconfigurable graph in hardware and active-inference memory form the optimal architecture for application level recognition systems like machine vision, anomaly detection, target recognition and the like. We also found this architecture to be well suited for continuous computation, which is the hallmark of recognition systems.

1.6.1 Structure of Recognition at Hardware Level

Graph-oriented architectures can deliver higher orders of magnitude higher performance on important graph processing algorithms than conventional, monolithic processor organizations and processor based systems. These graph machine architectures exploit high, parallel memory bandwidth to large numbers of small, fast memories and minimize node-to-node communication time; they can sustain their high performance per leaf-processing component when assembled into large collections of components.

As silicon capacities continue to grow, these architectures are an increasingly sensible way to turn silicon capacity into increased application performance, allowing us to economically solve modest problems quickly, and making larger and larger graph problems tractable. These Graph Machines can deliver new capabilities, with modest machines enabling real-time solution of hard

problems possible in deployed, operational scenarios, and large-scale machines supporting timely management and discovery of global-scale information.

Our FPGA-based Graph Machine implementation is able to perform better because of the high memory bandwidth [GraphStep06] and low Processing Element (PE)-to-PE latency. On a Virtex4-LX160-12, we are able to place 16 double-precision floating point PEs which operate at 285MHz each. This gives a raw floating point performance per chip of 9 double-precision Gflops which is higher than the Pentium-4 peak floating point performance at the same technology node (90nm). We store all data adjacent to the PEs in the embedded memory blocks so we can feed the PEs at their peak operating rate (Figure 6). A single Virtex4-LX160 can sustain 8.3 Gflops on Sparce Matrix Vector Multipllication (SMVM). For multiple-chip implementations, we use a low-latency time-multiplexed, butterfly fat-tree network to route among PEs. Using 32 leaf processing FPGAs (512 PEs), we are able to sustain a per leaf processing rate of 3 Gflops. More details on our first-generation FPGA-based SMVM implementation are reported in [DeHonDeLorimier05].

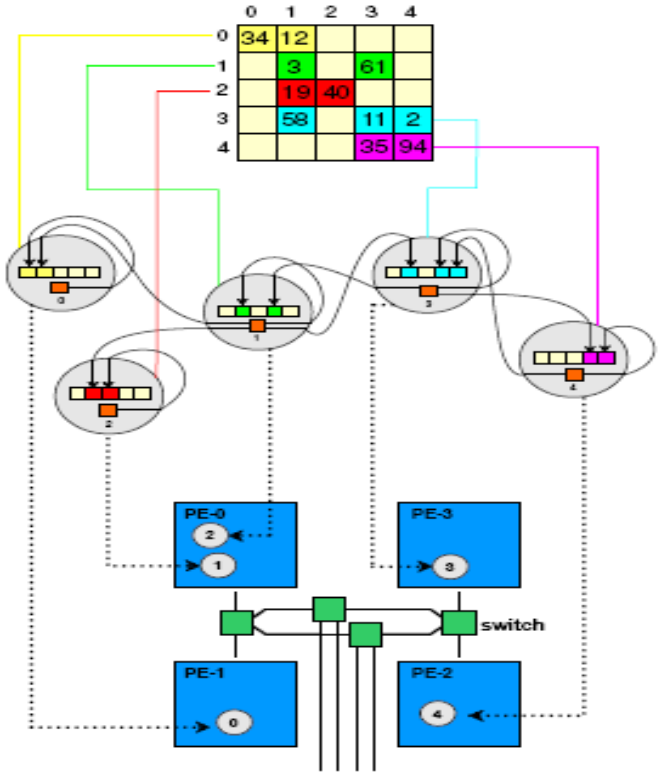


Figure 6. Assign Matrix Row Dot-Products (Graph Nodes) to PEs for Parallelism

1.6.2 Structure of Recognition at Application Level

During our investigation of low level recognition and its possible implementation at the hardware level, we faced the problem of granularity mismatch. Recognition at application level has a higher level of granularity and tends to be learning and matching intensive. And as such the memory operation become more complex and we transition into in PCAA terminology the micro

phase. This is a phase transition from recognition and perception to reasoning and inference. This is an indication that the critical resource limitation for this kind of recognition is not processing speed but rather memory density, bandwidth and structure. To investigate the impact of alternative inference-memory based architectures, we mapped an object recognition algorithm hybrid that combines Bayesian inference with the popular geometric hashing method [Wolfson97] to our proposed multi-core active-inference memory architecture. Geometric hashing is well-known in the computer-vision and medical imaging-research areas for its low-complexity and accuracy. The addition of heuristic Bayesian techniques, similar to those described in [Pfeffer01], makes the algorithm more tolerant of fuzzy objects such as distorted images or rotational and translational variants of the same image.

One obvious choice for implementing geometric hashing is the Content Addressable Random Access Memory (CA-RAM) architecture [Cho07], which places a hash function logic alongside conventional memory structures. Although this architecture can, in principle, be extended to perform the approximate hybrid approach, a natural extension to CA-RAM that incorporates approximate matching is the Programmable Object Evaluation Memory (POEM) architecture [Amduka06].

The POEM architecture matches stored content to an input feature vector (operand) through the user-definable distance function "F." Our first implementation of POEM on a Xilinx Virtex-2 has four, dual-port banks of memory, each 256 bits wide by 512 words deep. Each 256-bit row of memory represents eight integers of stored feature content. The four banks of memory are compared to new input feature vectors in parallel, according to the specified function, and the results are fed into a pipelined minimization tree. The data object associated with the best-responding row is returned.

The Traveling Salesman Problem (TSP) was chosen to test the recall ability and accuracy of such a memory. In learning mode, candidate problems are solved optimally, and the problem/solution pairs are stored. A replacement policy, which favors replacing least frequently used content with new inadequately represented content, flattens the response error.

1.7 Further Discussion

The results of these studies show that specialized hardware can significantly outperform general purpose processing (in some cases by orders of magnitude) for kernels of our chosen cognitive framework. However, along with Amdahl's law, there are additional obstacles to achieving system level performance on par with the speedups of the individual critical components. For example, the transformation of sensory data into a symmetry-invariant form suitable for lookup in a limited memory can be resource intensive and difficult to generalize.

In order to help system designers select cognitive processors, metrics are being developed. An indicator of performance identified here is "chunks-per-second," but in general, measures of performance of a cognitive system may not be so clear cut, and may not apply to different frameworks.

The architecture of cognition is a continuing research effort and the attempt to apply the general methods to this specific application should further extend our understanding of these frameworks.

The continued top-to-bottom implementation of applications using cognitive techniques should yield additional insights and capabilities. For example, while we have begun to study the kernels and cognitive frameworks in isolation, the integrated application should provide insights and opportunities, e.g., for specialized operators working within application-compatible reduced reliability or exploiting temporal or spatial locality in the control and data flow across kernels. The cognitive frameworks are, in a sense, interpreters running a high-level cognitive program; with more insight into the structure and behavior of this program we expect to be able to develop specialized system and hardware mechanisms to exploit them. System and hardware support that reduces the cost of using cognitive frameworks should facilitate further application of cognitive techniques, increasing the intelligence of important embedded systems.

2. Introduction

A hallmark of human cognition is its ability to find reasonable solutions to some very hard problems in the face of variations in the problems and in the surrounding context and to do so using proportionate resources. For example, salesmen routinely visit multiple US cities from one end of the country to another by finding reasonably good tours. Solving the same problem (aptly named the *Traveling Salesman Problem* (TSP)) by a computer, on the other hand, is known to be intractable for anything other than a modest number of cities. While progress is being made in solving TSP (and similar hard problems) [Applegate98], the progress is slow when measured against the resource-increments (both hardware and software enhancements) needed to solve them. As such, such problems (collectively called NP-Complete problems or, often, just "hard" problems) represent a fundamental challenge to our current computing methods and resources. A major characteristic of hard problems—their unscalability—is illustrated in Figure 7.

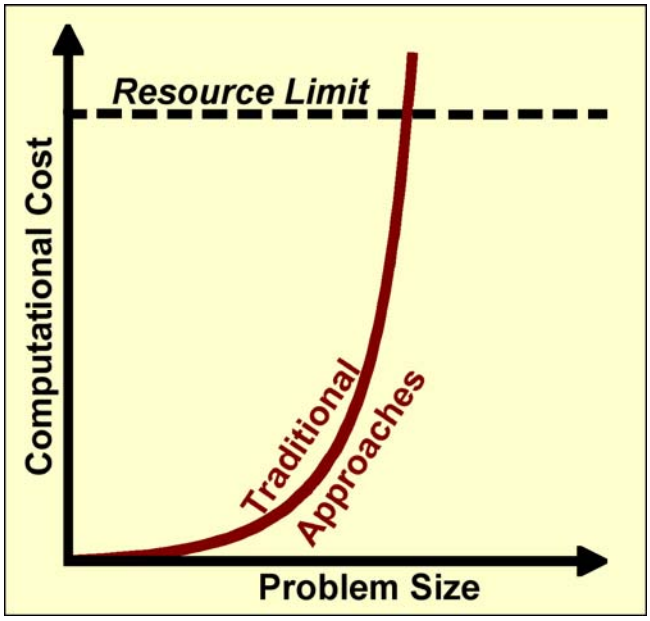


Figure 7. A significant characteristic of hard problems

Such hard problems occur, in one form or another, in everyday DoD activities such as mission planning for UAVs (includes aspects of hard problems such as bin-packing and TSP), and intelligence analysis (includes aspects of hard problems such as plan recognition).

We think that the current inability to provide reasonable solutions (produced by proportionate resources) to such hard problems stems not from any shortcomings in hardware speed or algorithmic shortcomings, but from something more fundamental: the basic architectural differences between the human cognitive architecture and the traditional von Neumann architectures currently in use for computers. As such, we propose a novel architecture—PCAA—that is inspired by (but not an emulation of) human cognitive architecture to find reasonable solutions for such hard problems at the cost of reasonable resources.

2.1 Goal of PCAA

A major goal of PCAA is to design a computer architecture to provide reasonable solutions for hard problems where those solutions are obtained:

- Across a variety of domains
- In a resource-efficient manner
- Through automated tradeoff of solution-quality and solution-cost
- Robustly in the face of variations in the problem or the context
- In anytime manner, with improved solutions over longer time

2.2 Background

2.2.1 Inspiration for PCAA Design

Human cognitive architecture (Figure 8) has been a major source of inspiration for PCAA since it is the only known general-purpose architecture capable of solving a variety of hard problems with a reasonable amount of resource in an anytime fashion.

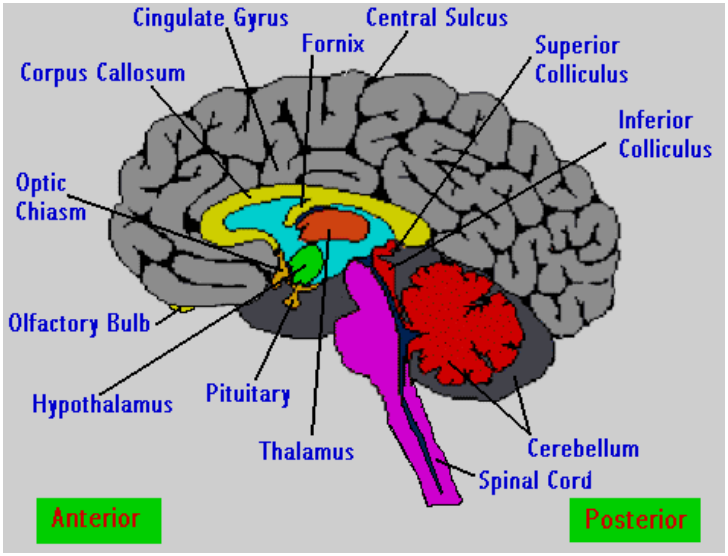


Figure 8. An overview of human cognitive architecture

Unlike computers, humans do not solve hard problems through exhaustive search or algorithmic computations. They use slow but massively parallel hardware to:

- Perceive the problem, represent it in structured form and generate perceptual solution constraints.
- Apply accumulated expertise to quickly generate local piecewise solutions and combine them into complete solutions.
- Apply general knowledge and reasoning ability to refine solutions and overcome holes in expertise.
- Improve their solutions iteratively and dynamically rather than strive for perfect, optimal first-time solutions.

2.1.1.1 Support for Linear Solving of TSP by Humans

Researchers found clear evidence to support the hypothesis that humans can provide reasonably good solutions to hard problems (such as TSP) in nearly linear time.

The basic Protocol used in this research involved an experiment where people were tested on a variety of planar TSPs of different numbers of points and varying shapes. Using a point-and-click computer interface, the solver was challenged to find the shortest path through the points, returning to the starting point, and doing so in the shortest amount of time possible (Figure 9). The goal of this research was to expose the process used by human solvers through collection and analysis of an extremely rich performance dataset. In addition, in some of the experimental manipulations, the impact of non-local problem features was explored by obscuring all of the display with the exception of the area directly around the mouse pointer. This forced solvers to use the mouse to reveal details of the problem they wished to consider, providing further insight into global and local processing used by solvers.

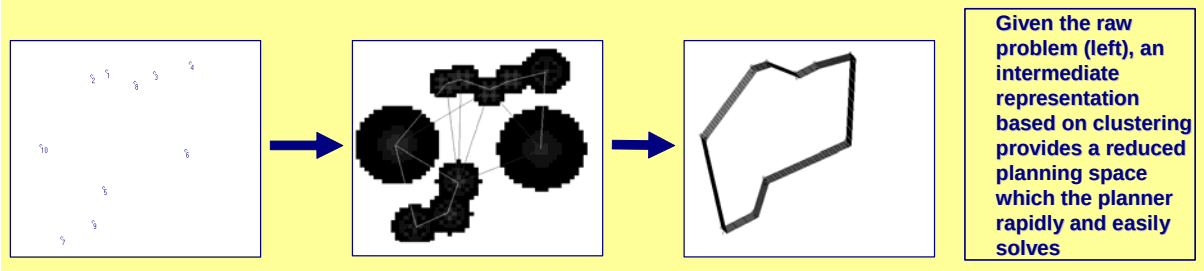


Figure 9. Support for linear solving of hard problems by humans

An analysis of the experimental results indicates that human approach for solving TSP is a combination of:

- Global parallel perceptual processes for grouping by proximity (problem decomposition).
- Perception of contours and goodness of path.
- Local serial search process.

The results from the experiments indicate that:

- Initial moves are more expensive than (constant) subsequent moves.
- Typical human solution within 5% of the optimal solution (good enough?).
- Profitably ignores non-local high-frequency problem features.

Thus, by transforming the problem representation, people solve a much simpler problem than computer scientists. This shows that if the perceptual representation is powerful enough, cognition can be relatively simple and provide good results.

2.3 Design Overview of PCAA Architecture

As hinted earlier, an architecture can solve hard problems only by simultaneously attacking the hardness of the problem from two sides (Figure 10):

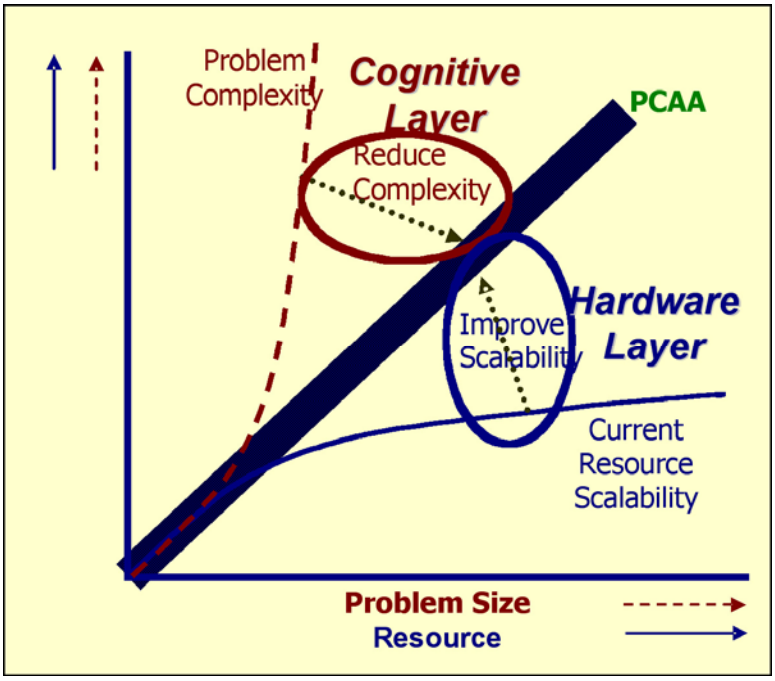


Figure 10. Major requirements for solving hard problems nearly linearly

- Reduce the algorithmic complexity – through a cognitive layer.
- Improve the hardware scalability – by designing an improved, more scalable hardware.

A cognitive layer can reduce algorithmic complexity through controlled, systematic cost/quality tradeoff. A hardware layer can provide highly efficient (e.g., parallel) implementations of algorithms, reduce communication overhead through improved processors, and improve the form-factor through reduced power consumption. The two layers can be combined through another layer (named Agent Virtual Machine (AVM) Layer in PCAA).

We think that an architecture to solve hard problems needs to synergistically integrate multiple approaches including: symbolic, subsymbolic, and hybrid representations; declarative and procedural knowledge; general knowledge and problem-specific knowledge; sequential and parallel computing; smart software and innovative hardware.

PCAA is an integration of three behaviorally orthogonal but functionally independent vertical components that include: the Cognitive Layer, the Agent Virtual Machine (AVM) Layer and the Hardware Layer (Figure 11). (An Application Layer can be built on top to complete a cognitive system, but is not part of the architecture itself.) In addition, the various layers are interfaced through appropriate languages: Cognitive Markup Language (CML) is created initially to interface between the Application Layer and the Cognitive Layer, and AVM Language to interface between the Cognitive Layer and the AVM Layer.

The major goal of the Cognitive Layer, as hinted earlier, is to reduce algorithmic complexity of hard problems. It tries to achieve this goal through intelligent collaboration of three major cognitive components (called “cognitions” or “cognitive levels”):

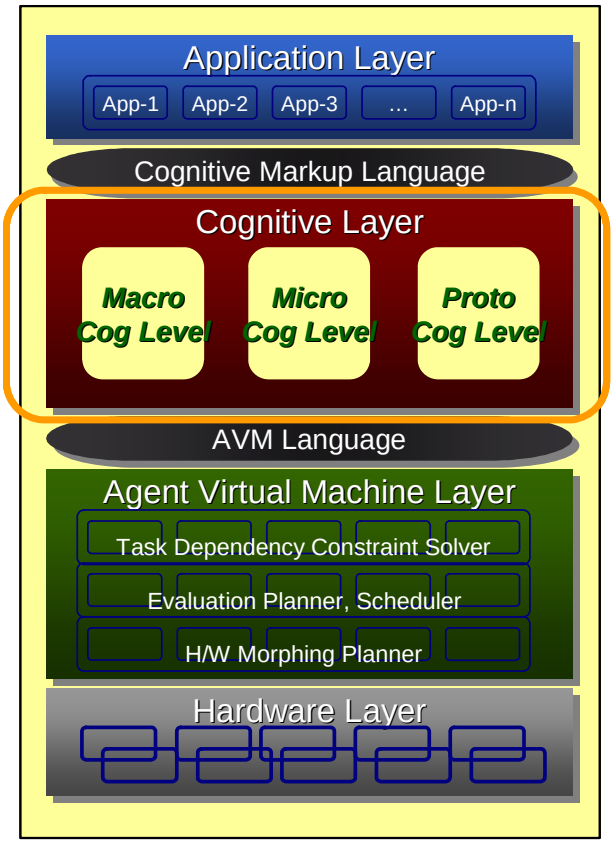


Figure 11. An architectural overview of PCAA

- Macro: It provides knowledge-based reasoning over symbolic representation. Based on Soar, it collects multiple reasoning approaches.
- Micro: It provides expertise-based reasoning over hybrid (i.e., mixture of symbolic and subsymbolic) representation. Based on ACT-R, it performs its reasoning through weighted associative matches.
- Proto: It provides the capability for distributed self-organization over subsymbolic representation. It performs its reasoning through swarming algorithms.

These cognitive components (i.e., Macro, Micro, Proto) interface with each other using a message-oriented language (called mCML) using XML syntax. The detailed design of the Cognitive Layer (including Macro, Micro, and Proto Cognitive Levels) is described in Section 3.4.

The major goal of the Hardware Layer is to improve the hardware scalability. It tries to accomplish this goal by providing hardware and parallel implementations of some essential cognitive algorithms (e.g., associative memory, Rete, swarming). In addition, the Hardware Layer incorporates some novel hardware component designs such as morphable processors. The major goal of the AVM Layer is to support the operations of the Cognitive Layer on the Hardware Layer by providing various functionalities (e.g., evaluation planner and scheduler, task dependency constraint solver).

2.3.1 Current Implementation of PCAA Architecture

The PCAA Cognitive Layer is currently implemented as a loose integration of three cognitive components (Macro, Micro, and Proto) interacting with each other using mCML-coded messages through an XML-Blaster-based message-server. The Cognitive Layer also interacts with the AVM/Hardware Layer and the Application Layer through XML-Blaster message-server (Figure 12).

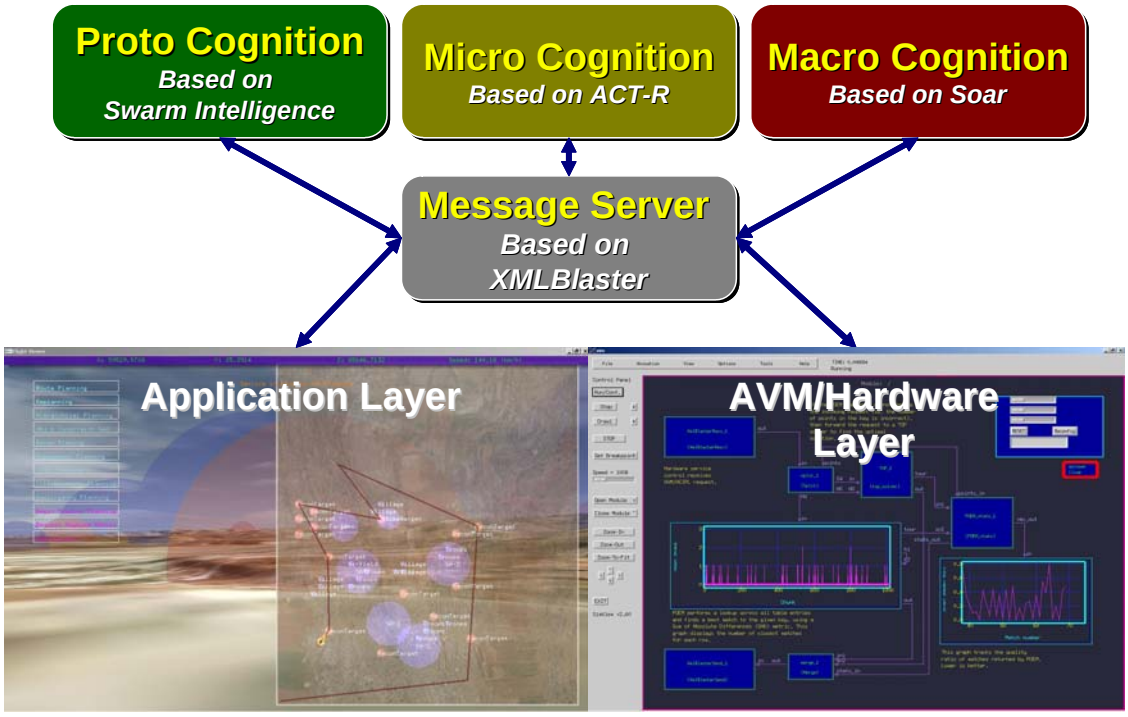


Figure 12. Current implementation of the PCAA architecture

2.3.2 An Overview of the Experimental Target Problems

We successfully applied this implementation to experimentally solve two very different kinds of experimental problems: an evidence marshalling application (details in Section 4.2.1), and a UAV mission planning experiment (details in Section 4.2.2). We present brief summaries of these problems below.

2.3.2.1 Sign of Crescent (SoC)

The Sign of the Crescent (SoC) problem is a problem created to help train human analysts at the US Joint Military Intelligence College. The SoC problem has the following features of interest:

- Involves learning to marshal dynamic, incremental evidences ("trifles") to recognize a coordinated terrorist plot.
- Evidences are scattered spatially and temporally.
- Problem includes distractors – intelligence reports that have nothing to do with the plot.

In the SoC scenario, three near-simultaneous terrorist acts are being planned:
- A dirty-bomb explosion aboard a ship in Boston harbor.
- A large bomb explosion aboard an Amtrak train (named "Crescent") in Atlanta.
- A large bomb explosion inside the NYSE.

The SoC problem is an instance of plan recognition: Infer planned intention(s) from observed relevant and irrelevant actions. In particular, it is an instance of Keyhole plan recognition – actors are unaware of (and neutral about) the plan recognition process. It is also a hard problem, with algorithmic complexity ranging from polynomial (top down) to exponential (bottom up).

2.3.2.2 UAV Mission Planning (UMP)

UAV Mission Planning (UMP), in general, is a large problem that potentially includes many tasks such as route planning, task allocation, scheduling, and collaboration. For our PCAA demo, we focused on a single-UAV route planning task. We developed a specific scenario for this task, called Lightning Bolt (described in Section 4.2.2). The Lightning Bolt scenario was refined down to a few simple, concrete components for use as a demo exercise by the Cognitive layer. These components were:
- Targets: A set of target locations was defined on the map. The UAV had to visit all targets, in an order of its choosing.
- Threats: A set of threat locations was defined on the map, each with a threat radius. The UAV had to complete its tour without entering any threat radius.
- Map: The scenario was defined to take place on a 2D map overlaid with a 100-by-100 coordinate grid. The positions of other components were specified in terms of this grid.
- UAV: The UAV itself was an implicit component of the scenario.

As defined, this scenario is very similar to the classic Traveling Salesman Problem (TSP). However, this problem has a strong cognitive aspect, and the goal was to focus on this aspect rather than to find a possibly quicker non-cognitive solution.

Additionally, the scenario was designed to support pop-up threats, which are threats that appear on the map after the UAV is in flight executing its original plan.

3. Methods, Assumptions, and Procedures

To design a cognitive architecture, we investigated many issues including:
- Requirements of a cognitive architecture including cognitive and hardware layers.
- Architectural principles.
- Design issues for a multi-level cognitive architecture.
- Design of a control system for the cognitive architecture.
- Design of a multi-level cognitive architecture.
- Metrics for evaluating a cognitive architecture.

3.1 Cognitive Architecture Requirements

PCAA cognitive architecture requirements span Cognitive Layer requirements and Hardware Layer requirements.

3.1.1 Cognitive Layer Requirements

The primary requirement of the Three Cognitions, One Integrated Architecture (C3I1) architecture is to efficiently solve hard problems that do not have straightforward or well-known algorithmic solutions. Those problems can be hard in the computational sense (e.g., NP-complete like the Traveling Salesman Problem), in the AI sense (e.g., AI-complete like the Turing Test), or some other way. "Solving" in our sense does not imply an exact solution; as real-world hard problems are often ill-defined and do not even have an exact solution, much less one that could be tractably derived. Our emphasis is on providing "good enough" solutions efficiently, while emphasizing other characteristics of the problem space and our solutions such as generality, taskability and efficiency.

3.1.1.1 Generality

Perhaps the most important requirement of our architecture is that it be general, i.e., applicable to a broad range of problems. The fields of computer science, artificial intelligence and machine learning often resort to a fundamental tradeoff between efficiency and generality of solutions, with the latter almost unerringly being sacrificed for the sake of the former. An extreme example might be Deep Blue, the special-purpose chess-playing computer that, after decades of effort, reached the goal of beating the World Chess Champion, only to make such extreme compromises in its approach (emphasizing search as its only organizing principle) and deployment (requiring special purpose hardware for move generation and such specialized domain functions) that it turned out to be useful for nothing else, even closely related games. Chess was meant to be a benchmark of cognition rather than an end in itself, but the focus on efficiency overtook any considerations of generality.

The primary constraint on our architecture is that it be applicable to a broad range of tasks along a number of dimensions, each with their implication for the architectural solution:
- Dynamism, i.e., the degree and rate of change in the problem as the environment evolves, requiring more incremental rather than batch solutions as change increases.

- Continuousness vs. symbolism, i.e., how much the problem is represented in continuous vs. symbolic terms, with broad implications in terms of representation and mechanisms.
- Determinism, i.e., how much moment-to-moment uncertainty is introduced by the environment. High stochasticity requires a broader examination of possible future states and limits the usefulness of methods such as search.
- Accessibility, i.e., how much of the problem environment can be known, requiring methods to deal with partial information, deception and uncertainty.
- Knowledge requirements, i.e., what needs to be represented and accessed dynamically vs. hardwired into an algorithm, with higher knowledge requirements implying more flexible control structures.

By *applicable*, we mean that C3I1 should provide the framework for a solution with little human adjustment or interference; otherwise the claim reduces to a relatively empty argument about computability. Instead, the architecture should provide the primitives to help solve the problems directly.

The solution should also be robust, in the sense of providing an acceptable, or smoothly degrading, solution as the problem definition changes rather than suddenly breaking down. This also implies that the architecture should be tolerant of incomplete, inaccurate or erroneous information. If an acceptable solution is not achievable, the architecture should recognize it and notify the user rather than return faulty results.

3.1.1.2 Taskability

An extension of the previous requirement is that the architecture should not only apply to a broad range of problems, but it should do so without requiring extensive modifications. This requirement carries a number of implications:

- It should provide anytime-solutions. If the architecture is interrupted while computing a solution (for example, to meet a real-time constraint), it should provide an approximate solution. The quality of the approximate solution is expected to get monotonically better with time. Moreover, provided ahead of time with time and/or quality constraints, the architecture should allocate its resources to provide the best possible solution given those constraints.
- It should be adaptive. It should handle novel variations of problems well. It should also improve with experience and learn from past solutions. It should adapt during a problem-solving episode to recognize changes in the environment and optimize its operations to those changes rather than repeat sub-optimal solutions.
- It should be extensible. The architecture should be modified or augmented with a reasonably small amount of human effort. Modification may be through direct change or through a programmability layer. The architecture should allow the adding in of new cognitive engines, such as new algorithms accomplishing the same functionality as existing ones, or special-purpose engines such as planners or optimizers. Extensibility also applies to adding new knowledge to the system and expecting the architecture to integrate it seamlessly in improving its performance rather than require reprogramming to modify previous knowledge to accommodate the new.

3.1.1.3 Efficiency

A key requirement of the architecture is the efficiency with which it will solve problems. Efficiency can be measured in a number of ways:

- Rapid response time. In many real-time domains, the speed of response is the key measure of efficiency. This requirement could take the form either of requesting the best response as rapidly as possible, or the best possible response within a fixed time constraint.
- Most efficient resource use. In other applications, such as embedded computing, resources (power, computing cycles, etc.) are severely limited. The measure of efficiency under those constraints is solution improvement as a function of resources expended.
- Scalability. Most solutions are tractable for some restricted range of problem complexity but quickly develop unreasonable time or resource requirements as the problem size grows and real-world complexities are factored in. A key measure of the efficiency of an architecture is the ability to keep its solutions tractable as the size and number of dimensions (and their interactions) of the problem grows.

3.1.2 Hardware Layer Requirements

The application analysis and mini-benchmark studies that we conducted led to the identification of a set of salient features that characterize algorithms used in cognitive information processing.

3.1.2.1 Dynamic Resource Management

Applications had highly variable resource requirements that varied over time and phases and across various interacting parts of the cognitive architecture. For example, the fine-grained concurrency that could provide dramatic speedup for swarming would not lead to a similar performance gain for an ACT-R or SOAR engine. Also, the importance of swarming versus SOAR or ACT-R varied over time as different parts of the cognitive runtime came into play to solve a particular task. Therefore, an efficient cognitive hardware architecture should support mechanisms for dynamic resource management.

3.1.2.2 Multi-granularity Parallel Execution and Data Transfer

The processing architecture must leverage parallelism at all levels of granularity. Proto cognitive tasks can benefit from extremely fine-grained reconfigurable logic via the asynchronous FPGA (AFPGA). Higher level cognition requires combinations of fine and coarse-grained processing. For tasks that cannot be mapped to the AFPGA because of size limitations, processor cores can implement fine-grained parallel execution via the communication network. An example of this type of task is finding the shortest path between two points in space.

The communication network provides messaging support, as well as support for interfacing to the external memory. A stream-based engine will be used to access off-chip memory to improve the bandwidth of data transfer from off-chip. The communication network can be implemented via conventional techniques, or slightly less conventional methods like high-speed serial links.

3.1.2.3 Data Tracking for Cognitive Applications

Cognitive applications are highly data driven. As data such as radar sensor readings flows through the system, it is important for cognitive applications to track where that data goes and how it gets used. Are routing decisions for a UAV being made primarily on the basis of a small number of sensors? The answer to questions like these are provided by tracking how data flows through the system.

The PCAA architecture will contain fine-grained data tracking. Software can identify a region of memory to the hardware, and the hardware assigns a token (number) to the region. The hardware tracks these tokens as data is used in computation. Software can then examine the token record to see what data was used for a particular result. The buffers that track the history of computation will be limited, and the exact management of them is a research question.

3.1.2.4 Power Management

A tile in tiled architecture, contains the three basic modules required to act like a stand-alone processor, namely an arithmetic logic unit (ALU), cache or local on tile memory and a communication switch. Each tile will have support for power management both directly and via the monitoring infrastructure. Performance feedback will be provided via monitors for run-time adaptation. The large on-chip memory will require power management as well. In particular, we will include support for "sleep" state memories, as well as deep sleep enabled by allowing the memory to lose state. This is useful when parts of the PCAA architecture are completely unused. Thermal management may not be necessary. However, we will provide on-chip thermal sensors that guarantee that the chip will never exceed its thermal budget.

3.1.2.5 Block Memory Transfer and Smart Memory Controller

One of the characteristics of ACT-R and other cognitive algorithms is that memory access patterns are deterministic. For instance, once a chunk is retrieved the next set of memory locations are provided by the pointers in that chunk. The memory controller must provide the ability to schedule memory transactions, and possibly provide functionality that permits pointer-chasing through data-structures in a user-programmable way.

3.1.3 Summary

The combined requirements of generality, taskability and efficiency seem impossible to meet. However, two key factors must be considered. First, as mentioned previously, the architecture does not precisely "solve" problems but instead provides the best possible solution given limited time and computational resources. Second, the architecture will be supported in meeting those requirements by special-purpose hardware that will implement key kernels of the architecture to achieve efficiency and scalability in their operations, and hence for the whole architecture. The ability of the architecture to decompose its operations into these fundamental kernels might be the key to achieving the target requirements. In addition, targeted hardware support (e.g., parallel

computing, dynamic resource management, efficient memory operations) is needed to realize this architecture.

3.2 Architectural Principles

The fundamental architectural principles underlying the C3I1 design are the cognitive pyramid, tight integration, and ongoing decision-making.

3.2.1 Cognitive Pyramid

The cognitive pyramid represents a tradeoff between the quantity of data being processed and the sophistication of reasoning applied to that data. Macro Cognition is powerful but computationally expensive, and so it is best applied to data that has been pre-processed and filtered by the other cognitive levels. In contrast, symbolic reasoning is much less natural for Proto cognition; however, Proto cognition's operations are simple, local, and easily mapped to massively parallel hardware, enabling it to handle large amounts of data. Micro cognition is a hybrid sub-symbolic/symbolic system and falls between Macro Cognition and Proto cognition in both power and scalability. These differences between levels naturally lead to the cognitive pyramid.

3.2.1.1 Semantic Data Pyramid

A simple instance of the optimal collaboration between the cognitive components is the formation of what we call a "semantic data pyramid" (Figure 13). In a semantic data pyramid collaboration, Proto processes the large amounts of input data to provide a structure on the input data. The much-reduced structured data is provided to Micro, which reasons over the data to try to solve the problems. Micro sends only a (relatively) miniscule amount of data to Macro—only the tiniest parts of the subproblems that it cannot solve itself. In a semantic data pyramid, the relationship between the various reasoning and various data-sizes among the cognitive components is as follows:

- Expensive reasoning over smaller problems: RMacro >> RMicro >> RProto
- Vastly reduced volumes of data for more complex reasoning: N >> N1 >> N2

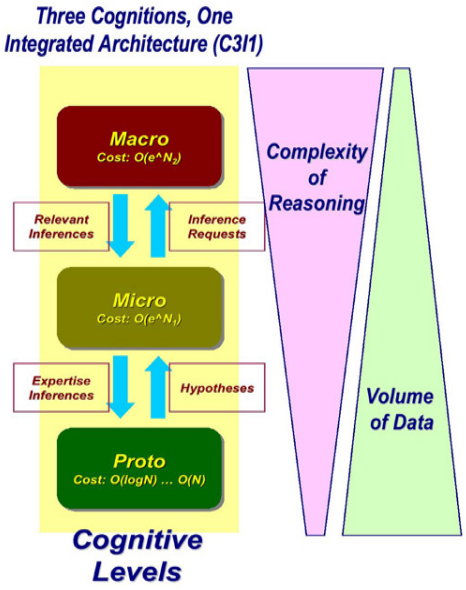


Figure 13. A collaboration of cognitive components forming a semantic data pyramid

3.2.2 Component Integration

A second fundamental architectural principle for C3I1the tight integration of its cognitive components: fine-grained full-story interaction between the cognitive levels or components. In addition, every information flow has a reverse learning flow. The cognitive components of the PCAA are tightly

integrated in memory, control, language dimensions to allow the entire system to be adaptive but focused toward a unique solution and exploit the optimum performance of each component with the minimum of computing resource requirement or complexity.

3.2.3 On-going Decision Making

When presented with a flow of information, C3I1 continually produces updated decisions, so that the system's results are any-time or in-time. If more accuracy is desired, additional time and resources can be allocated to allow more refined results; however, the current "best guess" solution is available at any point in the computations.

3.3 Design Issues for a Multi-Level Cognitive Architecture

There are a number of dimensions to the idea of integrating Swarming, ACT-R and Soar into a single homogeneous architecture. They are not quite orthogonal but they are independent enough that some could probably be pursued while others are not. Below is a brief summary of the dimensions, their pros and cons, and the anticipated difficulties in realizing the integration along that dimension both in terms of implementation and conceptually.

3.3.1 Loose Versus Tight Integration

Before going into the details of what an integration would involve, it is worth reviewing the general advantages and disadvantages of different degrees of integration. Some of these will be further specified below.

3.3.1.1 Advantages of Tight Integration

- A single model is conceptually easier to deal with than three separate models developed in their respective idioms and which will have to be reconciled as they are put together. The latter step is always harder than it looks, with many potential errors and mismatches between the respective conceptions.
- Based on our experience with ACT-R, which is itself a hybrid of symbolic and subsymbolic approaches, and general experience with hybrid cognitive systems, better leverage of the subsystems is usually obtained by tight integration where the systems are merged together rather than loose integration where they remain as separate subsystems. In the latter case, because so much information about each subsystem is trapped inside it and cannot be efficiently communicated to the other subsystems, many opportunities for synergy tend to get lost.
- It is much more efficient to develop a single model than three separate ones that then need to be integrated. Developing three models instead of one is not only inefficient because many things need to be duplicated, but integration is also time-consuming and introduces a loop at the end of the process that might cycle all the way to the start if significant conceptual changes in the individual models are revealed at integration to be necessary.

3.3.1.2 Disadvantages of Tight Integration

- Integration represents substantial work that might be better spent elsewhere, e.g., trying to squeeze more efficiency out of the independent cognitions, improving the applications, or speeding up the hardware implementation.
- Integration is a very slow process if done well. Integrated architectures like Soar and ACT-R represent decades of slow progress, putting mechanisms and representations together, seeing what works and what does not, and making subtle tweaks that turn out to make a significant difference. Integration is primarily a practical exercise that takes notable of experience to figure out. It is also slower and more complicated than a design document.
- Integrating the three cognitions together would mean losing the backward compatibility with the work developed in the original architectures. Some of it might still carry over to the new, integrated architecture, but it would almost come at the conceptual level and have to be adapted, rather than the more direct inheritance path that our loose integration can still claim at this point (e.g., Bob's use of the temporal logic implementation in Soar for the Feb 06 demo).

3.3.2 Shared Memory

This is a very important dimension in the integration, because having a separate memory for each cognition forces all partial results to be communicated back and forth, as is the case currently.

3.3.2.1 Advantages

- A common memory is much more efficient than passing partial results back and forth which in the case of our applications usually mean massive amounts of data. In some cases, such as Soar, returning a single inference instance to ACT-R, may not be a problem, but in other cases, such as Swarming passing the entire clustering results up to ACT-R, it could quickly prove unfeasible in real time for any sizable application. In many parallel hardware architectures, in fact, the systems become bound by the traffic being passed around rather than the local processing and minimizing this traffic becomes the key to successful implementations.
- Ensures consistency by making sure that all cognitions work from the same state of the system. While in theory the separate memories for each cognition will be synchronized to share the initial problem state and any updates, in practice discrepancies can crop up and spiral out of control.

3.3.2.2 Disadvantages/Difficulties

- The three cognitions currently have significantly different memory models. For instance, Soar makes a difference between declarative working memory and procedural long-term memory, whereas ACT-R has no separate working memory but has both declarative and procedural long-term memories. Also, swarm's decomposition of the problem might not correspond directly to the knowledge representation of ACT-R or Soar (e.g., in the UAV application), so what would their treatment by Micro/Macro Cognition be if they are part of a common memory?

- Going to a common memory model might be inefficient for the operations of each individual cognition. For instance, memory might not be organized in a way that makes most sense for the parallel swarming computations. Or the change in the memory model might negate the assumptions of Soar's RETE matcher. Or it might limit ACT-R's ability to implement a completely parallel memory matching. This depends on which model is adopted, but one has to expect the whole to be less efficient than the parts in their specific operations.
- The reimplementation cost would be significant. While the various cognitions are somewhat modular, changes to their memory system would have broad repercussions throughout much of their code and other modules.

3.3.3 Integrated Control

There are disagreements and uncertainties as to the level of control required between the three cognitions, but it seems reasonable to assume that some amount of control is required if they are going to be working efficiently with each other. They cannot just go and operate on their own without some constraints from the others. In a heterogeneous architecture, inter-cognition control might be as simple as cognitions being able to call each other, but could also involve outside control in Machine Cognitive Markup Language (MCML). Integrating either of these into a single internal process would seem like a good idea since neither of these look very satisfactory.

3.3.3.1 Advantages

- Integrated control would mean a single process that would call upon the primitives provided by the three cognitions. While there might be some advantages to self-organization, in the kind of applications that we are considering, it seems that strong control provides definite advantages in keeping the system focused upon its solution.
- Integrating control within the unified architecture would enable it to be more adaptive and generally more sensitive to the operations of the cognitions than control structures located either separately within each cognition or outside of the cognitions entirely.

3.3.3.2 Disadvantages/Difficulties

- We currently have little understanding which control primitives would be suited for the homogeneous integrated architecture. The basic concept seems to be an up-down loop where Proto cognition computes a first pass, Micro cognition attempts to apply its knowledge to the specific problem, Macro Cognition is called upon if that knowledge is insufficient, then results trickle all the way down to Micro and Proto Cognition and the cycle repeats. On the other hand, looser synchronization might also be possible where all cognitions can operate in parallel, but that might lead to dangerous conflicts and inconsistencies. There are also questions as to which cognition is best suited to perform what seems to be a general function of sequentially focusing upon each part of the problem (encountered both in SoC and UAV experiments).
- Because the control loop tends to sit on top of more basic mechanisms, it would probably not require too many or too deep changes to the various systems, but some of their basic assumptions about their course of processing might be challenged, e.g., what happens if an

operation that was viewed as unitary is now asynchronously interrupted by changes from another cognition?

3.3.4 Common Language

Each cognitive level is currently using its own language to specify models running in its architecture. Soar and ACT-R have relatively similar languages reflecting their common production system basis, but they do have significant differences. It is not clear how swarming models are specified, whether using a general-purpose computing language or a more specified language, but it is likely to be significantly different from a symbolic production system kind of language. Presumably the kind of information specified in defining a swarming model could be expressed in a manner similar to that of the activation calculus in ACT-R, though the ACT-R activation language is only used to specify values rather than arbitrary algorithms, as might be the case for swarming. This language would presumably be something like Cog-CML, but significantly more ambitious than its current conception.

3.3.4.1 Advantages

- A common language in which a single, integrated model could be developed would be much more efficient than having to specify three models, each in their separate language. Efficiency would not only be enhanced at creation (one would expect the single model to be close in complexity to the sum of the three separate models because much of the same information still needs to be specified), but especially at integration. Integration is a commonly neglected but often crucial part of the process. Creating three separate models would likely lead to many conceptual and programming errors that would have to be ferreted out at integration. This is often particularly difficult because different modelers usually write the separate models, and integration bugs often require their joint expertise. Writing a single model in a common language prevents many (possibly all) such integration bugs at the outset.
- A common language would also promote conceptual integration between the three cognitions by forcing them to express their concepts using a common lexicon. This forces the modeler(s) to think at the outset about the problem in terms of the integrated cognitions rather than each cognition separately. This in turn promotes better integration between the cognitions rather than trying to use them separately in ways that might not match well to their respective capacities.
- A common language would be much more accessible. We have not assumed that application developers would be required to write models for the cognitive level, but even if we assume that work (to whatever extent it needs to be done depending upon the reusability of the cognitive services) is reserved for specialized cognitive modelers, it is still a more likely to find (or easier to train) someone in this new language than require them to know ACT-R, Soar and swarming.

3.3.4.2 Disadvantages

- Inasmuch as it tries to gather most (possibly all) of the capabilities of the three separate cognitions in a single language, the result could be somewhat awkward, trying to be a jack-of-

all-trades and ended up doing nothing well. This can be alleviated if natural interactions between the three cognitions are discovered and a deep integration is achieved.
- The flip side of the previous disadvantage is that the deeper the integration, the more backward-compatibility with the original cognitive architectures is lost. A language that would preserve all of Soar, ACT-R and swarming might be possible but would probably be awkward. One that integrates them deeply would probably not look like any of them, and thus work developed in those architectures would at the very least have to be translated before being usable. Such a translation process might be accomplished automatically if direct enough, but our experience with that has met limited success.
- The reimplementation cost could be limited if there was a fairly direct translation from the new language to the original languages. Each system could remain relatively unchanged, with just an additional translation layer needed to generate the models in the original cognitions from the single model in the original language.

3.3.5 Integration Mechanisms

Probably the deeper and most difficult dimension is the integration of the architectural mechanisms themselves. It is also probably the most essential. Integration between swarming and ACT-R mechanisms would probably center on integrating the ACT-R activation calculus and the swarming output. That would mean making sure that the ACT-R calculus works well with the computations provided by swarming, and also figure out a way for learning in ACT-R to trickle down to swarming computations, e.g., through the pheromones. Integration between ACT-R and Soar based on the pattern of usage in the first application(s) would presumably center on impasses and chunking. ACT-R currently calls upon Soar when no relevant knowledge exists to apply directly, a concept very similar to the Soar impasse mechanism, something that is not an architectural primitive in ACT-R. The information returned by Soar is basically a summary of the result of its computations, including links from the problem-specific information furnished by ACT-R by removing the domain-general information used by Soar to arrive to its conclusion. This process is very similar to Soar chunking, and to some extent to ACT-R production learning, so the connection both ways seems like a good opportunity to integrate Soar capabilities into ACT-R. It is not clear exactly how Soar and swarming would connect at the mechanistic level, but some links would certainly be possible.

3.3.5.1 Advantages

- Integrating the mechanisms provided by the base cognitions provides the real leverage sought by putting them together. It makes each mechanism more powerful because it works with the other mechanisms rather than separately. The success of hybrid architectures often reflects their degree of integration. If the pieces are left relatively undisturbed, integration might still provide some benefits, but they are likely to be limited. Deep integration of fundamentally different mechanisms, e.g., the production system and Bayesian calculus in ACT-R, provides a system that provides more than just the sum of their capacities.
- Accomplishing the integration at the mechanism level once is much more efficient and productive down the road. Instead of having to figure out for every application or service how

to put the pieces together, that work can be done once and then assumed to work automatically. This is the main advantage of general architectures over specialized algorithms.

3.3.5.2 Disadvantages

- Deep integration takes a long time to figure out and is very difficult to achieve. The flip side of the generalization mentioned in the advantages is that the integration needs to be done in a very general fashion precisely because it is so broadly applicable. The new, integrated mechanisms need to be developed and made to work on a very broad sample of problems. It is not just a matter of hacking something for a one-off application.
- Again, and most deeply here, backward-compatibility with the original architectures is lost. For instance, you cannot take rules from a complex production system like Soar and assume that they will work in ACT-R because ACT-R puts restrictions on information access at the subsymbolic level, as a result of its integration with the Bayesian calculus. Conversely, you cannot take a Bayesian network and dump it into ACT-R, because ACT-R puts limit on the type and speed of inferences that can be accomplished because again they have to go through the other piece, i.e., the production system. The more one preserves the original power, the more one loses on the power of the integration. The key is to preserve as much of the original while getting most of the combination, but again that is a very long and tricky process.
- This kind of deep integration would almost certainly require a fairly complete reimplementation of each cognition, especially ACT-R. Changes in swarming and Soar might be more limited but are still likely to be significant.

3.3.6 Conclusion

A full integration of Macro, Micro, and Proto components would present significant challenges in both architecture and implementation aspects. Even if the abstract architectural issues were resolved its implementation will probably be unrealizable. So, it is particularly important to go after the most promising and the most feasible and realizable of this architecture. In particular the integrated control and common language dimensions of this architecture are the low hanging fruit. Common shared memory and integrated mechanism (cognitive algorithms) promise the most benefit but are difficult to generalize as a general cognitive information processing problem.

3.4 Design of the PCAA Cognitive Layer

The Cognitive Layer (Figure 14) includes three major cognitive components (called “cognition” or “cognitive level”):

- Macro: It provides knowledge-based reasoning over symbolic representation. Based on Soar, it collects multiple reasoning approaches. Because of the nature of its problem-solving strategy (exhaustive search), this process can be very expensive (exponential) though it can also provide very accurate solutions. This is described further in Section 3.4.2.
- Micro: It provides expertise-based reasoning over hybrid (i.e., mixture of symbolic and subsymbolic) representation. Based on ACT-R, it performs its reasoning through weighted

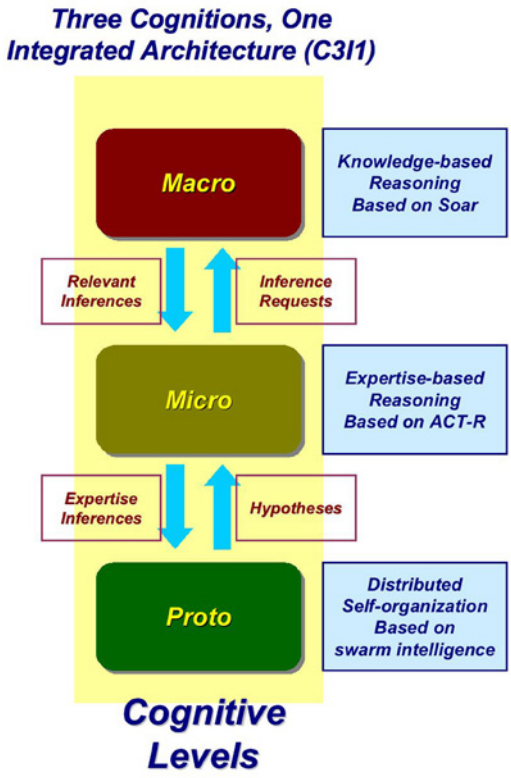


Figure 14. An overview of the PCAA Cognitive Layer

associative matches. The cost of this process is reasonable (nearly linear). This is described further in Section 3.4.1.

- Proto: It provides the capability for distributed self-organization over subsymbolic representation. Based on swarming algorithm, its reasoning process is cheap (sublinear to linear). This is described further in Section 3.4.3.

These cognitive components (i.e., Macro, Micro, Proto) interface with each other using a message-oriented language (called mCML) using XML syntax.

3.4.1 Macro Cognition

The primary task of Macro Cognition in the Cognitive Layer is to perform a deliberate "solution search" to solve a specific (sub)problem initiated by Micro cognition. The Micro cognition sub-problem may be incompletely/ill-defined. In essence, Macro Cognition provides brings knowledge to bear to deliberate over non-routine, novel tasks. Macro Cognition performs the problem search by bringing and integrating a range of different kinds of knowledge for the problem including:

- Application/domain knowledge.
- General types of inference (deduction, abduction, etc.).
- Relevance assessment of new inferences/directions of inference (a type of search control).
- Knowledge to modify/refine domain models/schemas.

3.4.1.1 Role

To perform problem search effectively and efficiently within the context of C3I1, Macro Cognition must satisfy a number of functional properties. The following outlines the functional requirements of Macro Cognition and the following section describes how Soar, the system used as the core component of Macro Cognition in the ACIP Phase I effort, meets the requirements and, as well, where additional functionality was needed to address the range of Macro Cognition's functional requirements. There are four primary requirements: ensuring Macro Cognition is responsive to the right problem, supporting multi-method reasoning, making knowledge search fast, and improving performance with experience (a kind of learning).

3.4.1.2 Computational Mechanisms

In Phase I of ACIP, we used the Soar architecture to realize the basic Macro Cognition component and built up some additional components on top of the Soar architecture to fully realize the requirements of Macro Cognition. This section describes how the Phase I system addressed the requirements described above; following sections lay out the implementation details and a summary of the complexity of Macro Cognition.

The Soar architecture was created to explore the requirements for general intelligence and to demonstrate general intelligent behavior [Laird87] [Laird95] [Newell90]. As a platform for developing intelligent systems, Soar has been used across a wide spectrum of domains and applications, including expert systems [Rosenbloom85] [Washington93], intelligent control [Laird91] [Pearson93], natural language [Lehman95] [Lehman98], and executable models of human behavior for simulation systems [Jones99] [Wray04]. Soar is also used to explore the integration of learning and performance, including concept learning in conjunction with performance [Chong05] [Miller96], learning by instruction [Huffman95], learning to correct errors in performance knowledge [Pearson98], and episodic learning [Altmann99] [Nuxoll04].

The Problem Space Computational Model (PSCM) [Newell91] defines the entities and operations with which Soar performs computations. Soar assumes any problem can be formulated as a problem space [Newell80]. A problem space is defined as a set of (possible) states and a set of operators, which individually transform a particular state within the problem space to another state in the set. There is usually an initial state (which may describe some set of states in the problem space) and a desired state, or goal. Operators are iteratively selected and applied in an attempt to reach the goal state. The series of steps from the initial state to a desired state forms the solution or behavior path. The steps Soar considers in finding a mapping from initial state to a goal state is the problem search in Macro Cognition. As discussed below, problem search is computationally expensive. One of the things that makes Soar powerful is the built-in mechanisms it uses to help alleviate the overall cost of problem search. These are sketched below and mapped to the requirements introduced in the previous section.

3.4.1.2.1 Macro Cognition Cycle of Operation

At a high level, many agent systems can be described by a sense-decide-act (SDA) cycle, as represented in the left of Figure 15. Soar's general processing loop, its *decision cycle*, maps directly to the SDA loop, as shown in the middle diagram. Individual components of the Soar decision cycle are termed phases. During the INPUT PHASE, Soar invokes the input function, communicating any changes indicated by the environment to the agent through the input-link. We developed message-passing infrastructure to allow Micro cognition requests to be communicated to Soar via the input-link. In the OUTPUT PHASE, the agent invokes the output function, which examines the output-link and executes any new commands indicated there, including responses for Micro cognition, which are packaged and sent out via output function processing.

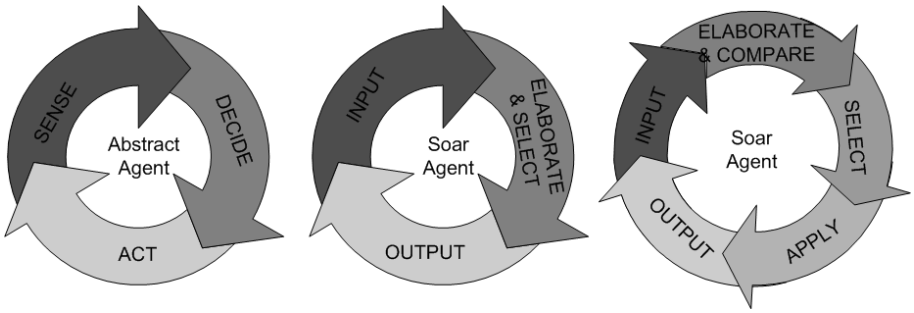


Figure 15. Common representation of an abstract agent as a cycle of perception, reasoning, and action (left), a high level view of Soar's sense-decide-act loop (middle), and a more-detailed Soar representation (right).

The reasoning within Soar's decision cycle is focused on the selection (and application) of operators. Each Soar decision consists of three phases within the "decide" portion of the SDA loop. During the ELABORATION PHASE, the agent iteratively fires any productions other than operator applications that match against the current state, including new input. This process includes "elaborating" the current state with any derived features, proposing new operators, and asserting any preferences that evaluate or compare proposed operators. This phase uses Soar's reason maintenance system to compute all available logical entailments (i.e., those provided by specific productions) of the assertions in the local memory.

When no further elaboration productions are ready to fire, the decision cycle is said to have reached *quiescence*. At this point, the elaboration process is guaranteed to have computed the complete entailment of the current state; any immediate knowledge applicable to the proposal and comparison of operators will have been asserted. At quiescence, Soar enters the DECISION PHASE and sorts and interprets available preferences for operator selection. If a single operator choice is indicated, Soar adds the operator object to memory and enters the APPLICATION PHASE. In this phase, any operator application productions fire, resulting in further changes to the state, including the creation of output commands. If there is not a unique selection for an operator (an *impasse*), Soar creates a subgoal, which allows the system to reflect on the impasse and propose other solutions.

3.4.1.2.2 Macro Cognition's processes

Within the decision cycle, Soar implements and integrates a number of influential ideas and algorithms from artificial intelligence. These processes define Soar's computational properties within the space of production systems generally and highlight how Soar addresses the requirements for Macro Cognition introduced previously.

3.4.1.2.2.1 Pattern-directed control

Soar brings to bear any knowledge relevant to the current problem in via associative pattern matching in a parallel match-fire production system. Thus, Soar systems determine their flow of control by the associations made to memory, rather than a sequential, deterministic control structure. Because the reasoning of the agent is always sensitive to the context, Soar readily supports both reactive and goal-driven styles of execution, and is able to switch between them during execution. Conflict resolution in typical Rule-based System (RBS) usually depends on syntactic features of rules. Soar uses no conflict resolution at the level of individual rules. Instead, conflict resolution occurs when choosing between operator candidates, allowing the decision to be mediated by available knowledge (in the form of preferences) rather than relying on syntactic features of the situation. This pattern-directed control is a key part of Soar's ability to provide **least-commitment control.**

3.4.1.2.2.2 Reason maintenance

Soar uses computationally inexpensive reason maintenance algorithms [Doyle79] to update its beliefs about the world. Every non-persistent object in Soar's memory is subject to reason maintenance, including impasses and operator selections [Wray03]. Reason maintenance ensures that agents are **responsive to their environments**. It also embeds knowledge about the dynamics of belief change in the architecture, with the result that agent developers are freed from having to create knowledge to manage revisions to current beliefs. This can greatly facilitate **multimethod reasoning** because the same approach to belief maintenance is consistent across all methods of reasoning.

3.4.1.2.2.3 Efficient pattern matching

Soar uses an extension of the Rete algorithm [Forgy82] to ensure efficient pattern matching across the entire knowledge base. A research demonstration showed that Soar can handle as many as one million rules without a significant slow down in reasoning [Doorenbos94]. By casting knowledge search as pattern matching between a Rete net of production conditions and asserted memory elements, Soar is able to realize **efficient knowledge search**, generally sublinear in the size of memory itself.

3.4.1.2.2.4 Preference-based deliberation

An agent in a dynamic environment must be able to deliberate and commit to goals. Soar balances automatic reason maintenance within the decision cycle with the deliberate selection of

operators. Because operator preconditions and actions components are implemented as separate rules, Soar agents recognize available options and reason about which option to take. This separation facilitates, at a low level, **least commitment representation** because there is no fixed, design-time mapping between the reasons for a deliberate activity and its implementation. This then also supports **multimethod reasoning** within Soar because the implementation of some specific activity (e.g., at a high level, this might include resolving a Micro cognition request) can be realized in many different ways.

3.4.1.2.2.5 Automatic subgoaling

In some cases, an agent may find it has no available options or has conflicting information about its options. Soar responds to these impasses and automatically creates a new problem space, in which the desired goal is to resolve the impasse. The agent can now bring new knowledge to bear on the problem. It might use planning knowledge to consider the future and determine an appropriate course for this particular situation. It might compare this situation to others it knows about and, through analogy, decide on a course of action. The full range of reasoning and problem solving methods available to the agent can be brought to bear to solve the problem indicated by the particular impasse, resulting in a powerful tool for facilitating the integration of **multimethod reasoning.**

3.4.1.2.2.6 Adaptation via generalization of experience

Soar performs knowledge search at the architecture level, using the Rete match process, while problem search engages both the architecture and the production knowledge representations. A fundamental assumption in Soar is that knowledge search should be made as inexpensive as possible, because knowledge search is an "inner loop" used in the problem search process. Soar's primary learning mechanism, chunking, converts the results of problem search within an impasse to new production representations that summarize the problem search that occurred within the impasse. This process results in new knowledge that will allow the agent to avoid a similar impasse and thus **improve performance with experience**.

3.4.1.3 Computational Structures

While Soar formed the core of Macro Cognition in ACIP Phase I, Soar alone does not constitute a complete solution for Macro Cognition. At a minimum, Soar would need to be configured to look for and accept problem requests from Micro cognition, choose between different available reasoning methods, and need encoding of domain specific information. Figure 16 illustrates the conceptual architecture of Macro Cognition. There are five major components:

1. Soar, the implementation language for the other components and the base architecture for Macro Cognition, as described above
2. Control knowledge, which is used to dynamically guide problem search within Macro Cognition
3. Domain knowledge, which is brought into Macro Cognition via an automated process and used in problem search

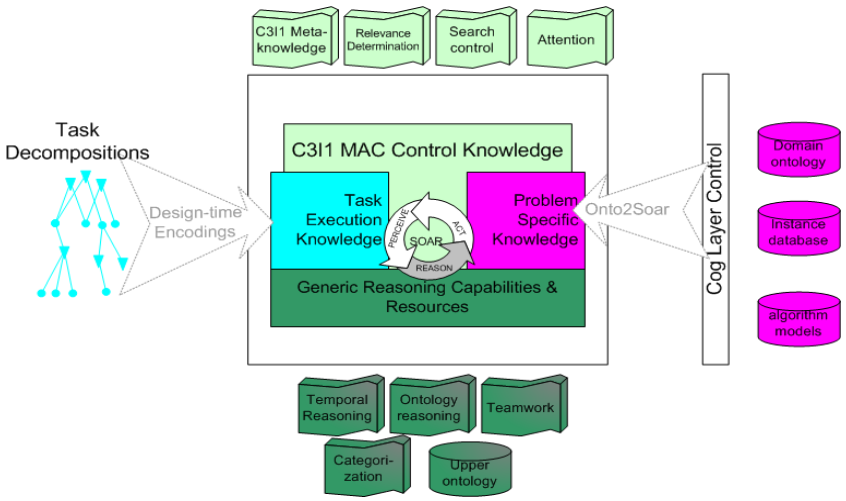


Figure 16. Conceptual architecture of Macro Cognition for the C3I1 architecture.

4. General reasoning capabilities, which are existing libraries of reasoning methods that can be applied to particular problems
5. Task execution knowledge, which will include any domain-specific reasoning processes that must be encoded for a particular domain

While we expect that reasoning systems other than Soar could be realized at the core of this architecture, the additional components would be needed regardless of what reasoning system was placed at the core of Macro Cognition. The following subsections outline each of the non-Soar-specific components in more detail.

3.4.1.3.1 Control Knowledge for Macro Cognition

General knowledge is needed to set up a "problem" in Macro Cognition, assess progress and likelihood of fruitful lines of reasoning, and to determine which knowledge areas should be applied in the exploration of a particular problem. For example, in our demonstrations (described below) knowledge was encoded to interpret messages from Micro cognition and send responses in the required format back to Micro cognition.

For general realization of Macro Cognition, the control processes will need to be significantly expanded. The most important issue is developing solutions to address the potentially very large size of the search space. Given any single problem, reasoning could proceed in many different directions. Further, given any point in time, many different active queries might be active. Another challenge is recognizing when some particular request has been satisfied. There are thus at least three distinct categories of knowledge within this general area of control:

- **Attention**: At any one time, there may be many active queries within Macro Cognition. Attention knowledge helps prioritize queries according to measures such as urgency, importance, and simplicity (e.g., it might sometimes make sense to interrupt reasoning and resolve a simple [few step] query. Attention is distinguished from relevance estimation primarily by attention's concern with distinguishing between multiple queries rather than assessing progress towards the resolution of any individual query.

- **Relevance Estimation**: The goal of Macro Cognition's reasoning is to find a relationship (possibly the best relationship) between two previously unconnected objects in the C3I1 memory. Reasoning costs must be directed effectively in order to provide an overall computationally efficient approach to this goal, because the branching factor is likely to be very large. Heuristic knowledge must be developed to provide intermediate indications of progress, to assess possible "progress," and to learn from previous experiences how to organize and rationalize reasoning. There is no simple "Manhattan distance" heuristic that can be used to assess whether some intermediate reasoning step is semantically "close to" some desired result. Thus, some additional research is needed to evaluate intermediate results and direct the search for new values.
- **Problem/Query Resolution**: In the best case, Macro Cognition will find *relevant* relationships that can link two or more items specified in the Micro cognition query (i.e., the focus of attention). However, simply finding a relationship is no guarantee that link is important in the context of the problem. For example, in the Sign of the Crescent scenario, in response to a query about finding a relationship between Abu al Masri and Hans Pakes, one possible link is that they are Arab. While true, it's not really the important inference (which is that the two names refer to the same person). Thus, Macro Cognition will need knowledge to assess the likely importance/relevance of any link that is found.

In ACIP Phase I, we encoded hand-crafted solutions for this class of knowledge, sufficient for the purposes of demonstrating the interaction of Micro cognition and Macro Cognition and allowing Macro Cognition to actually execute the general problem search. For the future, significant research will need to be focused here to develop algorithms that can effectively guide problem search in very open-ended problem domains.

3.4.1.3.2 Domain Knowledge (Ontologies, Instances, Algorithms)

Some representation of the domain is needed by Macro Cognition. We assumed in Phase I that a general domain representation would be available via the application layer. In general, there will be three classes of domain knowledge:

1. Domain Ontologies. These are formal descriptions of the terms used in the domain. In the Sign of Crescent problem, there might be an associated "terrorism ontology" defining specific relations in the terrorist domain. Example: "a terrorist is a member of a terrorist organization." Ontological domain knowledge can be expressed using the general approach outlined for general ontology reasoning (below).
2. Domain facts/instances. These are the facts representing the problem and data associated with problem. Examples: "al Qaeda is a terrorist organization," "Osama bin Laden is a (known) terrorist." One role of C3I1 is to increase the size of this instance knowledge base, so that future instances of similar problems will draw from increasingly more broad collections of knowledge. General instance knowledge is expressed in C3I1 memory, rather than directly to Macro Cognition.
3. Algorithms. Algorithms are used to specify specific ways in which content should be interpreted/manipulated. In Phase I, we assumed application knowledge instances could be derived solely from ontological and instance classes.

In addition to domain specific ontology knowledge, the system may need to obtain additional ontological knowledge. Additional knowledge will be especially needed for high-level concepts not likely to be explicitly represented in the domain (e.g., formal definition of an organization). These concepts are typically defined in an upper ontology, such as the Standard Upper Merged Ontology. Upper ontology representations can be directly coded into Macro Cognition.

In addition to an upper ontology, the system may also need to draw on other ontologies for the definitions of terms not defined by the application layer. For example, the Sign of the Crescent includes the notion of an "alias," yet it's unlikely that the term "alias" would be represented in a (narrowly construed) terrorism ontology. Increasingly, it is becoming possible to go the World Wide Web and locate Ontologies that describe or define terms (lexical databases, such as WordNet, also ease dependence on specific terms, allowing a system to interpret thesaural relations between items). For example, the system might note that Abu al Masri, a person of interest, is associated with Hans Pakes via the undefined "alias" relation. This might trigger a search to understand the "alias" relation, allowing the system to "understand" that "aliases are a kind of also-known-as relation," and then, using this new understanding, create new domain facts/instances dependent on that understanding ("Abu al Masri uses the alias Hans Pakes").

However, in the long term, this approach is not a viable solution. Instead, the system will need the ability to fill in gaps in its ontological knowledge, which will require knowledge to seek out other ontologies, to validate their accuracy, and then incorporate that new information with existing knowledge.

For Phase I, we assumed any ontological information needed was represented and developed hand-coded ontologies sufficient for the demonstrations. Automated translation of ontologies in the OIL language to Soar is available via Soar Technology's proprietary tool, Onto2Soar.

3.4.1.3.3 Generic Soar Reasoning Components

General methods of reasoning are necessary to support the open-ended reasoning methods required of Macro Cognition. General methods from previous Soar systems that were used in ACIP Phase I include elements drawn from deductive inference, planning, temporal reasoning, spatial reasoning, and ontological reasoning (as discussed above).

3.4.1.3.4 Task Execution Knowledge

The general reasoning components outlined above all served, in ACIP Phase I, the task of answering a Micro cognition request. However, in the course of our demonstrations, there was some over-arching procedure that served as a model for the functioning of the architecture for that task. In Sign of the Crescent, this procedure was evidence marshalling, in UAV Mission Planning, it was the notion of state-space planning. Thus, in Phase I, we decomposed these "Generic Tasks" into a number of constituent elements and mapped them onto individual pieces of cognition. For example, Macro Cognition's role in the general problem of UAV mission planning was solving route planning problems with a small number of points. In Phase I, these "generic task" elements were all identified via manual analysis and created by hand. In the

medium-term, this kind of mapping will likely continue to be necessary. However, as more capabilities and libraries are developed for each library, it may be possible to generalize the specific implementation components for each task such that this component of the Macro Cognition architecture is subsumed by the other application-neutral components.

3.4.1.4 Hardware Realizations

Macro Cognition is based on SOAR. The core production matching of SOAR targeted for hardware implementation is based on an algorithm known as Rete [Forgy82]. Rete is an exact-match algorithm that uses two memory types: the *working memory* contains facts about the world that are collected over time, and the *production memory* contains rules. Each rule in the production memory would typically be a set of conditions, and a set of actions to perform if those conditions are met. An added complexity is that the condition expressions may contain variables that must be bound during matching.

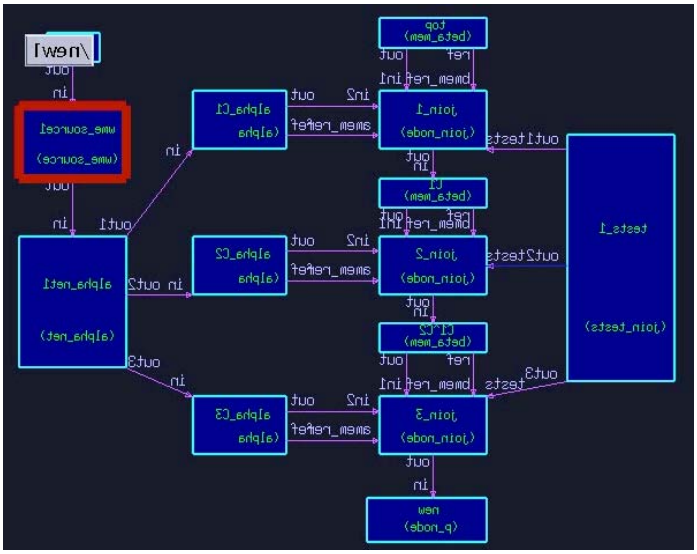


Figure 17. Rete in event-driven logic

The study of Rete hardware targets a simulation of the event driven processing afforded by asynchronous logic [Manohar04]. Figure 17 shows an asynchronous data flow network for a Rete, where each node is built from fine-grained logic, and can execute concurrently as soon as dependencies only when necessary reduces energy consumption for are met. As is typical with asynchronous approaches, computing computation, and performance can be greater by removing the need for a central clock.

Due to the dependencies of typical networks, the maximum concurrency achievable is only on the order of 10X to 20X. Hardware support for hash functions yields higher constant factors of acceleration and are under consideration. The library used to exercise the Rete experiment is shown in appendix A.3.

3.4.2 Micro Cognition

3.4.2.1 Role

Micro cognition's role in the Cognitive Layer is to store and deploy problem-solving expertise. A key part of the ability to solve complex problems efficiently is in learning from experience and then deploying that knowledge as a substitute for the computations, trial-and-errors and other time-intensive processes by which we have accumulated it. One can view that process as a space-time tradeoff, where we are trading off increased storage space for expertise in exchange for

reduced processing time in finding an expert-level solution. This relies on the assumption that storage is getting increasingly cheaper and that special-purpose hardware can provide storage with the right properties (robust associative content-based retrieval, constant time access, etc.) while basic computational limitations combine with the time-critical nature of many applications domains to make time a non-negotiable constraint.

Micro cognition performs its function by storing in memory specific patterns that are likely to be useful in future problem-solving episodes, then retrieving them and deploying them to bypass future computations. Those patterns can take the form of declarative knowledge holding the solution to various sub-problems or procedural skills providing more efficient computational shortcuts. This knowledge can originate from a number of sources both inside and outside of the architecture:

- It can be provided by the user through high-level languages such as CML. This knowledge engineering approach is limited by the form of knowledge that can be extracted from human users and formulated symbolically. Not all expertise fits that description, as the experience with expert systems has well established.
- It can result from reasoning and inference processes at the Macro Cognitive level, which flows top-down to Micro cognition after inadequacies in the Micro cognitive expertise knowledge base are discovered and Micro Cognition requires assistance from Macro Cognition.
- It can result from bottom-up regularities discovered at the Proto cognition level that are formalized in terms of subsymbolic knowledge at the Micro cognition level. This expert knowledge is often considered to be implicit, and thus cannot be extracted and formalized easily because it is neither consciously known nor completely symbolic in nature.

3.4.2.2 Computational Mechanisms

Micro Cognition is defined by basic steps of sequential cognition that rely on a hybrid mix of symbolic and statistical knowledge, such as applying a set of patterns to the current situation to make the best guess under tight computational constraints. The computation is a mix of sequential and parallelizable, similarity- and history-based and a mix of bottom-up and top-down constraints. Micro Cognition is composed of declarative and procedural components. Key operations include retrieving declarative knowledge from long-term memory, and selecting the most promising production rule from its procedural skill set. Those operations are a combination of symbolic pattern-matching and subsymbolic processes such as spreading action and similarity-based partial matching, together with learning processes such as Bayesian learning for declarative memory and reinforcement learning for procedural memory.

3.4.2.2.1 Spreading Activation

Spreading activation is a mechanism operating at the subsymbolic level of declarative memory. Its role is to guide the retrieval process to the most likely pieces of information by statistical means. While rule-based systems typically emphasize logic-based matching where the symbolic pattern precisely defines the pieces of information to be retrieved, that is seldom the whole story in real world domains. Experts accumulate very large stores of information (e.g., chess experts have been estimated to learn up to millions of chunks of knowledge over their careers), so

narrowing down the search for the right one(s) is essential to computational efficiency. But finding the right one is seldom a matter of logical matching, since a key aspect of intelligence problems is their under-constrained nature. Instead, experts are able to focus on the right few pieces of information without quite being able to explain why, indicating that while the actual knowledge is often explicit (and thus able to be extracted and engineered into expert systems), the ability to retrieve it is implicit, and thus statistical in nature [Wallach00]. At the Micro cognition level, that process is implemented in the spreading activation mechanism that propagates the activation that drives retrieval of information from the current problem focus to relevant pieces of information in the expertise knowledge base. The strengths of associations driving the spreading activation process result from computations at the Proto cognition level that reflect factors such as compatible structures between problem definition and expert solution.

3.4.2.2.2 Partial Matching

Once a piece of expert knowledge such as an inference or a previous solution pattern has been retrieved, it needs to be compared to the actual problem to determine if it is indeed applicable to the problem description since the statistically driven retrieval process is only guaranteed to return knowledge that is likely but not certain to be relevant. Since real world domains tend to be complex, approximate and uncertain, the current problem will almost never match exactly existing expertise resulting from previous experiences. Instead, what is needed is a way of determining what the closest match is and whether it is good enough to provide a solution. That is the role of the partial matching mechanism that uses similarity-based generalization to find the closest match to an item in memory. That generalization capability is similar to that provided by neural networks and other distributed representation mechanisms that provide statistical rather than logical generalization. While the pattern driving the matching is described symbolically, which enables structured reasoning processes by the architecture, its internal mechanism generalizing that pattern is statistical, again achieving a synthesis between symbolic information and subsymbolic mechanisms.

3.4.2.2.3 Bayesian Learning

Real world domains tend to follow general statistical patterns such as the tendency for information more frequently or recently accessed to generally be of greater use. This principle is quite general and present in many domains, and underlies some of the most popular applications in the information domain such as web search engines. While these patterns can be re-learned on a domain-by-domain basis, it is more efficient to embed them into the architecture so they can be leveraged more efficiently [Lebiere03]. Micro cognition accomplishes that goal by attaching to each piece of expert knowledge activation quantities that will combine with the previously described spreading activation quantities to drive information retrieval. Those quantities are learned using scalable, online Bayesian algorithms.

3.4.2.2.4 Reinforcement Learning

The mechanisms previously described all apply to the retrieval of declarative information. Micro cognition also includes a procedural memory that stores expert skills rather than knowledge.

While information retrieval is driven by factors such as recency, frequency and similarity, procedural skill acquisition and tuning is controlled by success and failures in accomplishing goals. Similarly to activation quantities for declarative knowledge, utility quantities are associated with procedural skill to control their selection and application. Those subsymbolic utility quantities are learned using scalable, online reinforcement learning algorithms.

3.4.2.3 Computational Structures

At this point, from the entire modular architecture of ACT-R we will focus on two modules: **declarative memory** and **procedural memory** (a.k.a. the central production system). These are the most longstanding, best-defined and most functionally rich modules of the architecture. Other architectural modules are either significantly under-specified (e.g., the goal module basically defaults to the goal buffer since the goal stack and associated computations were eliminated), functionally irrelevant to the basic task of this project (e.g., the motor module is basically concerned with the human factors aspect of keyboard and mouse manipulation) or both (e.g., the perceptual modules are primarily concerned with a theory of sequential attention and do not incorporate elaborate functionality for pattern recognition in the vision and auditory systems). Moreover, those are the two modules that will benefit more directly and thoroughly from parallelism and other hardware layer functionality. We will detail below the functional aspects of those modules relevant for an efficient, parallel implementation at the hardware level, and how they could be framed into kernels.

3.4.2.3.1 Declarative Memory

Declarative memory is organized as a set of chunks that are named structures with a (usually small for reasons of cognitive plausibility) number of fields, each of which can contain another chunk (thereby making possible arbitrarily complex hierarchical representations) or a literal value usually taken directly from perception or some other basic process. The primary operation for declarative memory is the retrieval of a single chunk by **matching of a retrieval pattern** (i.e., a partial chunk description) against the set of all chunks in memory. Barring an additional complication discussed below, the pattern can be matched against each chunk independently, thereby providing perfect parallelism for memory retrieval as a function of the number of chunks.

This process seems trivial enough as described above. It is the purely symbolic aspect of memory retrieval. However, in general multiple chunks will match the retrieval pattern, or none will match it exactly. The process therefore includes a second level, the subsymbolic level that computes for each chunk a match score based upon the chunk's **activation** and its degree of match to the pattern. Without going into the details of the activation and matching equations at this point, the activation of a chunk is the sum of a base-level activation component (which reflects the history of use of the chunk), a spreading activation component (that represents the priming of each chunk from context elements such as the components of the current goal) and a stochastic component (that is simply noise designed to make the system non-deterministic). The **match score** of the chunk is then computed by subtracting from the chunk's activation a sum of penalties that reflect the degree of mismatch between each retrieval pattern component and the

actual chunk content. The degree of mismatch is computed from similarities between chunks and between literal values that are similar to distributed representations in neural networks. Overall, these computations reflect Bayesian statistics designed to make the system adapt to the statistical structure of its environment and capture the sort of effortless (as opposed to symbolic cognitive) implicit information processing that underlies much of human cognition. Again, those computations can be performed in parallel for each chunk, and are usually the most computationally demanding aspect of performing memory retrievals.

Each of the components of the subsymbolic activations computations are **learned** using Bayesian estimation to reflect the statistics of the environment as well as the statistics of use of memory. Thus the base-level activation increases with new references (retrievals or re-creations of the chunk) and decreases with time. Spreading activation reflects the statistics of co-occurrence between context elements and specific memory chunks. Stochastic noise varies for each retrieval attempt. Similarities between chunks can be learned by a process similar to Latent Semantic Analysis to reflect the similarity of their contents. All these processes potentially apply to each chunk at every step of cognition, again making parallelism a major computational advantage. Chunks themselves are also learned from the contents of buffers such as the goal, but this isn't a heavy computational process. However, this might have an impact on how chunks are distributed across processors.

There are a couple of complications to this process. First, declarative memory is segmented into chunk **types** associated to particular slot values. Types can be organized in (single-inheritance) hierarchies similar to many object hierarchies. When a retrieval pattern is processed, only chunks of the type specified (and subtypes) qualify for matching. This is not fundamentally important but it might have an impact on the distribution of chunks on a parallel architecture, e.g., scattering chunks of the same type across processors. The second complication is a process called **blending** which, instead of retrieving a single chunk, retrieves the best consensus chunk from all candidates, weighted by match score. The first part of the matching process is still the same, but there is a second re-combination part at the end that might be tricky to parallelize well. Similarly, chunks holding similar contents could also be merged to preserve whatever built-in memory limitations the hardware system might carry with limited degradation in performance.

3.4.2.3.2 Procedural Memory

Procedural memory is composed of a set of condition-action rules. The conditions do not match the entire contents of modules such as declarative memory or the visual field but instead just the content of a buffer associated to each module (e.g., the last chunk retrieved from memory and the visual percept currently being focused upon). Conversely, the action side of a production consists in requiring actions from some module(s), such as sending a retrieval pattern to declarative memory or shifting attention in the visual field, that in turn result in a change in the associated buffer. This organization makes the primary operation of procedural memory, the **matching of production rules against the current buffers** and the selection and firing of the best matching rule, a highly parallelizable process as well, because each rule can be matched independently against a small set of information and the kind of complexities such as a full unification process used in more complex production systems are here avoided. In that way, the process is highly

similar to retrieval from declarative memory, with the production rules taking the place of chunks and the buffer contents serving the same selection role as the retrieval pattern. Thus the matching process for procedural and declarative memory could very well be generalized to a single process applicable to both memories, except for the fact that the subsymbolic levels are quite different (though could potentially be partly unified, but it would require substantial changes) and that modulates the symbolic matching through processes such as partial matching.

As for declarative memory, multiple productions can match at any given cycle, or none might match exactly, requiring a subsymbolic process called conflict resolution to arbitrate between production rules and select the winner to be fired. The key concept here is not activation but **utility,** which is a combination of probability of the rule leading to a successful solution times the value of the goal, minus the future costs of the solution (usually defined in terms of time). As for activation, the probabilities and costs are learned according to a reinforcement learning-type mechanism.

A number of qualifications are similar to those for declarative memory. First, only productions matching the type of buffer contents (especially the goal type) are considered for matching. Second, the utility value of a production can also reflect the degree of match to the buffers, using the same similarity-based computations as those used for partial matching in memory retrieval. Third, the utilities have varied in their degree of goal-specificity, a distinction similar to flat versus hierarchical reinforcement learning. Finally, a process to learn productions also exists, but might be too complex and not parallelizable enough to be worth worrying about at this point.

3.4.2.4 Hardware Realizations

The primary building block required of the hardware layer by ACT-R is the provision for storage and parallel matching of declarative memory chunks and production rules. Because these processes are highly parallelizable, considerable savings could be realized that would make the system largely insensitive to the size of its knowledge bases, which would make great strides toward achieving linear performance as a function of problem size. These processes could be viewed anywhere from four separate kernels to a single one, with the most likely outcome being two separate kernels for declarative and procedural memory. We will review the alternatives here briefly:

- Four kernels: these would consist of one kernel each for the various combinations of declarative/procedural and symbolic/subsymbolic. The main argument for this division is that it would allow for possible sharing of kernel functionality with other levels, especially Macro Cognition.
  - — Declarative symbolic: storage and matching of chunks, basically a conventional associative memory. This kernel could be unified with the similar functionality at the Macro Cognition level, though subtle but substantial knowledge representation differences exist between the two.
  - — Declarative subsymbolic: learning (largely Bayesian) and computation of statistical quantities controlling access to declarative chunks.
  - — Procedural symbolic: storage and matching of production rules. This kernel could be unified with the similar functionality in Macro Cognition, except that the production rules

at the Micro cognition level are considerably more constrained and thus might benefit from a more efficient implementation that exploits those constraints.
  - Procedural subsymbolic: learning (roughly reinforcement learning) and computation of utility values determining selection of production rules.
- Two kernels: these would consist of one kernel each for declarative and procedural memory. Basically, this would correspond to the proposal above, with integrated symbolic and subsymbolic functionality to maximize efficiency. Given the tight integration between symbolic and subsymbolic information in Micro cognition and the way these mechanisms impact each other, it probably doesn't make sense to implement them as separate kernels. The downside is that those kernels are probably not reusable at the Macro Cognition level.
- One kernel: since many of the mechanisms, especially symbolic pattern-matching, are quite similar between declarative and procedural, it would seem possible to unify them and provide the entire functionality in a single kernel. However, substantial differences would remain (e.g., declarative knowledge flexibility of access versus condition-action rule asymmetry of application) that the cost would probably be significant in terms of efficiency, and the single kernel would most resemble a main switch with heterogeneous sub-kernels.

It is worth noting that the brain does not have separate areas for symbolic and subsymbolic information (differences in generalization and representation between some regions notwithstanding) but it does provide separate physiology for declarative knowledge and procedural skills. This would support the two-kernel route, which is the most likely at this point.

3.4.3 Proto Cognition

3.4.3.1 Role

The Micro and Macro Cognition levels in C3I1 are powerful but computationally expensive. In order to solve hard problems in linear or sub-linear time using the C3I1 architecture, it is necessary to reduce the complexity of the raw data from n to n_1, where $n >> n_1$. This is the role of the Proto Cognition level in the Cognitive Layer. This general goal may conceivably be accomplished in a variety of ways; the current design envisions two specific functions: 1) organize the problem into structured, hierarchical representation for processing by Micro and Macro Cognition, and 2) provide skeletal subsolutions for those upper levels. Together, these two processes reduce the search space and provide a hierarchical, searchable database for the upper levels. In addition to this bottom-up process, Micro and Macro Cognition will guide Proto cognition's activities from the top down, either directly or indirectly. In ACIP Phase I, the technology used for Proto Cognition has been *swarming*, or *stigmergic Proto Cognition*. Currently, there are two specific mechanisms, or *kernels*, used in Proto Cognition: *SODAS*, for decentralized hierarchical clustering, and *marker-based stigmergy*, for routing. The next section first describes swarming in general, and then each of the kernels in more detail.

3.4.3.2 Computational Mechanisms

3.4.3.2.1 Swarming: Stigmergic Cognition

"Stigmergy" is a term coined in the 1950s by the French biologist [Grassé59] to describe a broad class of multi-agent coordination mechanisms that rely on information exchange through a shared environment. The term is formed from the Greek words *stigma* "sign" and *ergon* "action," and captures the notion that an agent's actions leave signs in the environment, signs that it and other agents sense and that determine their subsequent actions. Different varieties of stigmergy can be distinguished. One distinction concerns whether the signs consist of special markers that agents deposit in the environment ("marker-based stigmergy") or whether agents base their actions on local components of the current state of the fulfillment of the system's goals ("sematectonic stigmergy"). Another distinction focuses on whether the environmental signals are a single scalar quantity, analogous to a potential field ("quantitative stigmergy") or whether they form a set of discrete options ("qualitative stigmergy"). As shown in Table 1, the two distinctions are orthogonal.

Whatever the details of the interaction, examples from natural systems show that stigmergic systems can generate robust, complex, intelligent behavior at the system level, even when the

Table 1. Varieties of Stigmergy

	Marker-Based	Sematectonic
Quantitative	Gradient following in a single pheromone field	Ant cemetery clustering
Qualitative	Decisions based on combinations of pheromones	Wasp nest construction

individual agents are simple and individually non-intelligent. In these systems, the most important locus of intelligence is not in a single distinguished agent (as in the centralized model) nor in each individual agent (the intelligent agent model), but in the interactions among the agents and the shared dynamical environment.

An example of stigmergic cognition builds digital analogs of the pheromone fields that many social insects use to coordinate their behavior. We have developed a formal model of the essentials of these fields, and applied them to a variety of problems. Different "flavors" of pheromones can record different kinds of information, and different species of agents can make use of different combinations of flavors in their decisions.

The real world provides three continuous processes on chemical pheromones that support purposive insect actions.

- It aggregates deposits from individual agents, fusing information across multiple agents and through time.
- It evaporates pheromones over time. This dynamic is an innovative alternative to traditional truth maintenance in artificial intelligence. Traditionally, knowledge bases remember everything they are told unless they have a reason to forget something, and expend large amounts of computation in the NP-complete problem of reviewing their holdings to detect

inconsistencies that result from changes in the domain being modeled. Ants immediately begin to forget everything they learn, unless it is continually reinforced. Thus inconsistencies automatically remove themselves within a known period.
- It diffuses pheromones to nearby places, disseminating information for access by nearby agents and integrating local information into a global pattern.

We model these dynamics in a system of difference equations across a network of "places" at which agents can reside and in which they deposit and sense increments to scalar variables that serve as "digital pheromones," and these equations are provably stable and convergent ([Brueckner00]). They form the basis for a "pheromone infrastructure" that can support swarming for various C4ISR functions, including path planning and coordination for unpiloted vehicles, and pattern recognition in a distributed sensor network.

Stigmergic mechanisms have a number of attractive features for military systems:
- **Simplicity -** The logic for individual agents in stigmergic systems is much simpler than for an individually intelligent agent. This simplicity has three collateral benefits.
  1. The agents are easier to program and prove correct at the level of individual behavior.
  2. They can run on extremely small and simple platforms.
  3. They can be trained with genetic algorithms or particle-swarm methods rather than requiring detailed knowledge engineering.
- **Scalable -** Stigmergic mechanisms scale well to large numbers of entities. In fact, unlike many intelligent agent approaches, stigmergy requires multiple entities to function, and performance typically improves as the number of entities increases.
- **Robustness -** Because stigmergic deployments favor large numbers of entities that are continuously organizing themselves, the system's performance is robust against the loss of a few individuals. The simplicity and low expense of each individual mean that such losses can be tolerated economically. When domain entities are few in number, we typically use "ghost agents:" each domain entity continuously generates alternatives and self-organizing to maintain a dynamic plan that the higher-level entity then follows.
- **Environmental Integration -** Stigmergic techniques facilitate integration with the environment in two ways, structural and dynamic.
  — Structurally, stigmergic mechanisms exploit the topology of the environment (Table 2). For instance, pheromone mechanisms store information in a distributed fashion, close to where it is generated and as well as close to where it is needed. In many cases, the environment provides its own model of its structure, greatly simplifying the modeling effort within the intelligent system. To a first approximation, one can think of a stigmergic system as a distributed blackboard, where the distribution reflects the intrinsic structure of the problem domain.
  — Dynamically, stigmergic systems directly integrate environmental dynamics into the system's control, and in fact can enhance system performance. (The dynamic nature of the environment is an important distinction between stigmergic systems and blackboards.) A system's level of organization is inversely related to its symmetry, and a critical function in achieving self-organization in any system made up of large numbers of similar elements is breaking the natural symmetries among them (Ball96). Environmental noise is usually a threat to conventional control strategies, but stigmergic systems exploit it as a natural way

Table 2. Some Application Domains of Swarming

Domain	Project (Program)	Topology	Agents	Emergent Structure	Conventional Integration	References
Resource allocation	AARIA (DARPA Agile in DSO); AORIST (DARPA ANT in IXO)	Timeline (total order); Eligibility matrix of tasks and resources (bigraph)	Tasks; Resources	Schedule	Heuristic scheduler produced by ISI	Brueckner and Parunak, 2003
Robotic path planning and C2	ADAPTIV (DARPA JFACC in ISO); ERA (DARPA NA3TIVE in IPTO)	Discretized battlespace (manifold)	Threats, targets, neutral entities, vehicles under control	Paths	Soar (supports Tac-Air Soar in JSAF)	Parunak et al., 2002
Sensor Networks	internal	Nearest-neighbor network of microsensors (quasi-manifold)	Microsensors	Presence, location, velocity of target		
Team formation for collaborative sensing in UAVs	LEN (DARPA WASP in IXO)	Nearest-neighbor network of UAVs (quasi-manifold)	UAVs	Specialized team to image or radiolocate a target	Conventional centralized signal and image processing algorithms	Odell, et al., 2003
Biosurvei-lance	internal	Network of point-of sale terminals (scale-free network)	Cooperative pattern recognizers	Location and characteristics of threat		Brueckner and Parunak, 2002
Network management	CASANDRA (DARPA Knowledge Plane in IPTO)	Internet (scale-free small world)	Network management agents	Varied	Conventional learning and planning technologies (various)	
Information extraction	Ant CAFÉ (ARDA NIMD)	Ontologies (scalefree small-world)	Linguistic relations	Substantiated concept maps	Reinforcement learning of analyst hypotheses and priorities (Sarnoff Corporation)	
IED threat prediction	STIFLE (ONR CIED)	Road network (quasi-manifold)	Convoys, patrols, insurgents	Location of IED threat	Statistical reasoning	
Urban battle prediction	STRONG ARM (DARPA	Discretized battlespace (manifold)	Threats, targets, neutral	Enemy actions and battle	Soar and statistical reasoning	

Domain	Project (Program)	Topology	Agents	Emergent Structure	Conventional Integration	References
	RAID)		entities, vehicles under control	outcomes		
Distributed hierarchical clustering	SODAS (ARDA AFE)	Cluster tree	Documents, clusters	Cluster tree		Parunak, Rohwer, et al., 2006

to break symmetries among the entities and enable them to self-organize. In addition, close coupling to environmental dynamics enables stigmergic systems to provide any-time processing.

Swarming is a general subsymbolic paradigm that has been successfully applied to many different domains. Currently, swarming is implemented in two kernels for the C3I1 architecture: distributed hierarchical clustering using SODAS (an instance of sematectonic stigmergy), and route finding using marker-based stigmergy. The following sections describe these kernels.

3.4.3.3 Computational Structures

Proto cognition applies SODAS (Self-Organizing Data and Search) structure to organize data. SODAS is a swarming variant that implements dynamic decentralized hierarchical clustering, to maintain a large, dynamic, hierarchically-structured database of targets for efficient access by Micro and Macro Cognition. In SODAS, the nodes represent either data nodes or summaries of subtrees of data nodes, and the links represent parent-child relationships between the nodes, so that the network forms a hierarchical tree of data clusters (Figure 18). Fundamental sematectonic operations on the nodes are Merge, where a node merges two of its children that are especially similar to each other, and Promote, where a node promotes a child that is especially distinct to be a sibling. If search, or foraging, is performed on the tree, then each node may also contain a set of pheromone strengths, to aid marker-based foraging agents in their search.

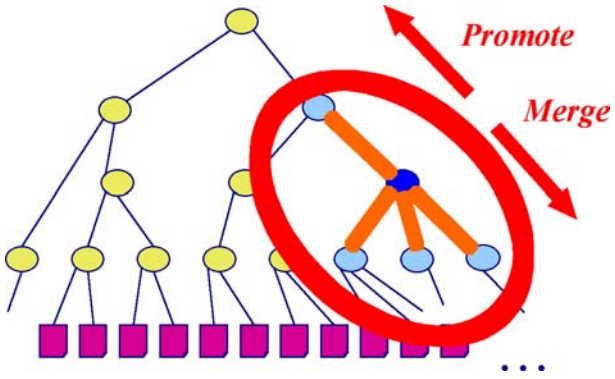


Figure 18. The network architecture of SODAS. Here each document or cluster is a node, linked to its parent and its children, forming a cluster tree. Promote and merge are basic operations on the tree, and search agents may search the tree for documents or clusters, depositing pheromones on the way down the tree.

3.4.3.3.1 Comparison with other Clustering Methods

Clustering is a powerful, popular tool for discovering structure in data. Classical algorithms [Jain99] are static, centralized, and batch. They are *static* because they assume that the data and similarity function do not change while clustering is taking place. They are *centralized* because they rely on data structures (such as similarity matrices) that must be accessed, and sometimes modified, at each step of the operation. They are *batch* because they run their course and then stop.

Some applications require ongoing processing of a massive stream of data. This class of application imposes several requirements that classical clustering algorithms do not satisfy.

Dynamic Data and Similarity Function - Because the stream continues for a long time, both the data and users' requirements and interests may change. As new data arrives, it should be able to find its way in the clustering structure without the need to restart the process. If the distribution of new data eventually invalidates the older organization, the structure should reorganize. A model of the user's interest should drive the similarity function applied to the data, motivating the need to support a dynamic similarity function that can take advantage of structure compatible with both old and new interests while adapting the structure as needed to take account of changes in the model.

Decentralized - For massive data, the centralized constraint is a hindrance. Distributed implementations of centralized systems are possible, but the degree of parallel execution is severely limited by the need to maintain the central data structure. One would like to use parallel computer hardware to scale the system to the required level, with nearly linear speed-up.

Any-time - Because the stream is continual, the batch orientation of conventional algorithms, and their need for a static set of data, is inappropriate. The clustering process needs to run constantly, providing a useful (though necessarily approximate) structuring of the data whenever it is queried.

Clustering algorithms fall into two broad categories [Jain99]: those that partition the data (such as K-means) and those that yield a hierarchy. We wish to construct and maintain a hierarchy, to enable efficient search. [Gordon96] offers a detailed survey of five approaches to this task.

The most common approach is *agglomerative* clustering, which repeatedly identifies the closest two nodes in the system and merges them until the hierarchy is complete.

Divisive algorithms begin with all leaves in a single cluster. At each stage a cluster is selected and divided into subclusters in a way that maximizes the similarity of the items in each cluster.

Incremental algorithms add leaves one by one to the growing structure. This approach supports a certain degree of dynamism in the data, but in work up to now, the emergent structure is strongly dependent on the order in which the leaves are presented, and cannot adjust itself if the distribution of data changes over time.

Optimization algorithms adjust the hierarchy's structure to minimize some measure of distance between the similarity matrix and the cophenetic distances induced from the hierarchy. Current versions of this approach are centralized.

Parallel methods attempt to distribute the computing load across multiple processors. [Olson95] summarizes a wide range of distributed algorithms for hierarchical clustering. These algorithms distribute the work of computing inter-object similarities, but share the resulting similarity table globally. Like centralized clustering, they form the hierarchy monotonically, without any provision for leaves to move from one cluster to another. Thus they are neither dynamic nor any-time.

SODAS includes both agglomerative and divisive operations. It is incremental in allowing continuous addition of data, but in the limit the structure that emerges is independent of the order in which leaves are presented. It can be interpreted as a hill-climbing optimizer, and runs on multiple processors, without the requirement for centralization that constrains previous work. Furthermore, the quality of SODAS clustering increases exponentially over time (Figure 19). The current retrieved can be retrieved at any time it is needed; if more refined results are desired, additional clustering can be performed to achieve them.

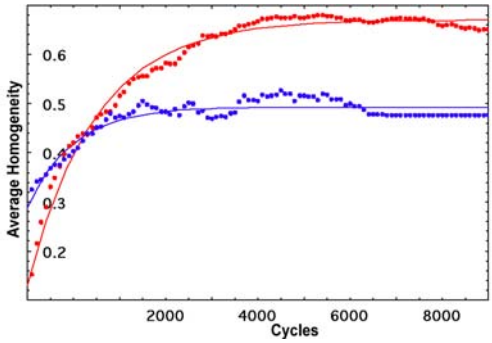


Figure 19. The quality of SODAS clustering increases exponentially over time

3.4.3.3.2 Marker-Based Stigmergy

Marker-based stigmergy depends on special-purpose information (such as pheromones) that agents deposit in the environment to support coordination. An example in nature is ant foraging, where ants that have found food lay pheromones down as they return to the nest, which attract other ants to the food source (Figure 20).

This process is simple, distributed, robust, and dynamic, which distinguishes it from many other routing algorithms. In the context of C3I1, it is used to provide skeletal solution frameworks to the higher cognitive levels for such problems as UAV routing.

In this case, the nodes are spatial locations, or *place agents,* and each node has links to each of its spatial neighbors (for instance in a 2D lattice). Swarming agents roam over this spatial topology, reacting to pheromone concentrations and other agents, and depositing their own pheromones.

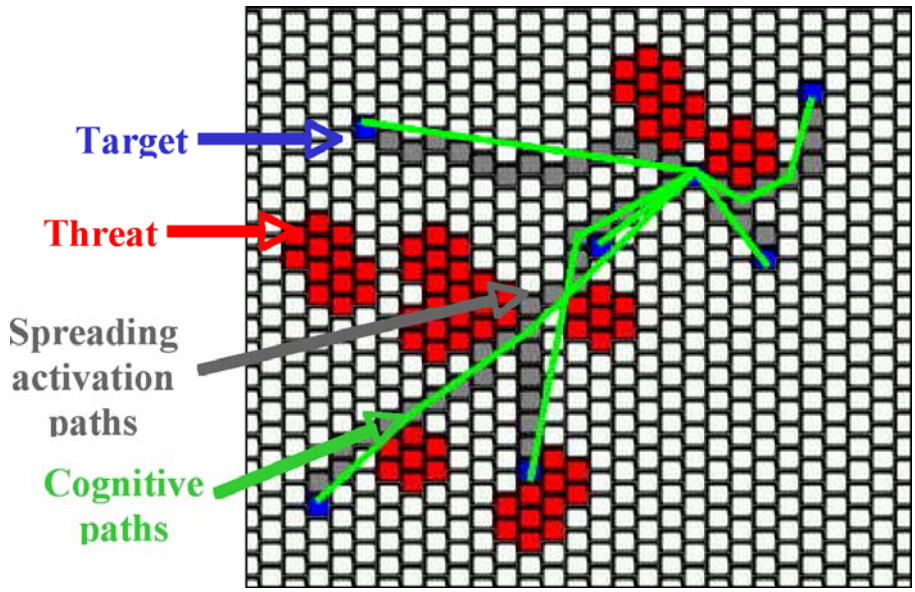


Figure 20. Identifying best paths in threat-warped space using marker-based stigmergy. Here each spatial location in the lattice is a node, connected to its neighbors by links, and agents roam across the lattice, depositing pheromones.

3.4.3.4 Hardware Realizations

The swarming mechanisms used to implement Proto Cognition in C3I1 have the advantage of being easily implemented in massively parallel hardware. The mechanisms are naturally distributed and local, with little need for the storage or broadcast of global information. Thus it should be possible to attain near-linear speedup in the number of hardware nodes.

One long-term platform option is asynchronous programmable logic (APL). Simple state machines and local state would be mapped entirely onto APL, with more complicated behavior spilled to a microprocessor. The swarming algorithms are loosely synchronized by nature, making them especially amenable to APL implementation. Furthermore, APL only uses dynamic energy during updates, and swarming nodes are designed to be dormant by default, unless they need to be updated.

An example FPGA implementation of SODAS is sketched out in Figure 21. The basic cycle is as follows:

1. Select a “crow’s foot” from the node database
2. Assign each of the elements of the crow’s foot to a separate FPGA circuit
3. Perform the node computation
4. Store the elements back into the node database

To avoid conflicts, the nodes drawn from the database may be locked. Alternatively, conflicts may be resolved simply by letting the last database write decide the issue: As long as the structure remains a valid tree, the algorithm is designed to tolerate errors and be self-correcting, since it is inherently stochastic.

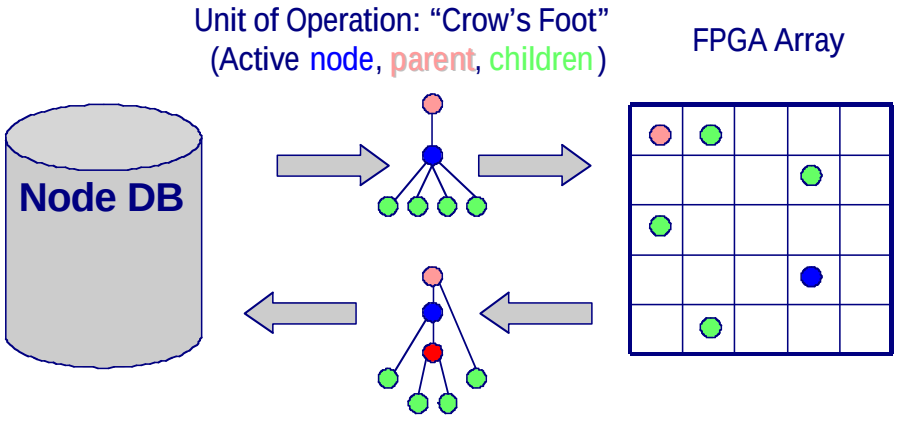


Figure 21. A possible FPGA implementation of SODAS clustering

3.4.4 Control and Synchronization of Cognitive Components

3.4.4.1 Approach

The problem of control in general, and resource allocation in particular, is primarily defined by the information that has to be taken into account to make (near) optimal decisions and only later by the kind of algorithm used to implement it. Both the nature of the cognitive layer, with three cognitions attacking the same problem at different levels, and the problems it tackles, with parallel improvements possible over the entire problem structure, suggests a model of constant competition for resources by each cognition (Proto, Micro and Macro). The basic question is to which cognition to allocate the next chunk of computational resources to obtain maximum incremental improvement in solution quality. This need not be winner-take-all but could result in an uneven distribution over the various cognitions.

Let us frame this problem in the domain of evidence marshalling (i.e., SoC). At a given point in time, the cognitive system is attempting to perform an inference. Let us assume that Proto has produced a clustering of the current trifles and Micro has attempted without success to match *n* existing inference instances to the top-rated cluster while Macro hasn't yet been called upon. The possible options for allocating the next chunk of processing are as follows:

1. Micro: attempt to match another inference instance
2. Proto: improve clustering and call back Micro
3. Macro: attempt to reason about this cluster and solve inference
4. Give up: abandon this inference and move on to next cluster

For each option, the decision should be based on past experience. Relevant statistics for each option include:

1. Probability that next inference instance will be the right one for this cluster (probably as a function of number of past instances attempted since this should decrease with activation rank ordering) vs. computational effort to match and instantiate it.
2. Probability that further clustering will improve the probability of finding the right inference (which depends upon how many cycles have already been expended, and ultimately in turns

depends upon Micro and/or Macro) vs. computational effort to run Proto clustering for additional cycles. This is not an all-or-none decision but rather a matter of how many more cycles to allocate.

3. Probability that Macro will have the right knowledge to reason and find the right inference for this cluster (not clear on what basis anything more than a base rate could be computed, but Macro might have additional insights in its own operations) vs. computational effort to attempt reasoning. Additional benefit of providing a new inference instance to Micro is difficult to quantify.
4. Probability that a useful inference for this cluster cannot be found (as a function of past efforts expanded on it at every level as well as other characteristics, e.g., current quality of cluster in terms of coherence and such) vs. expected computational cost to solve the problem without this step (or probability that abandoning this step would lead to ultimate failure in solving the problem).

For each option, the relevant statistics are basically a probability that allocating resources to this option would lead to finding a solution vs. the expected computational costs of pursuing this option. The weighing of probability vs. cost might in turn depend upon parameters passed to the cognitive layer from the application layer, e.g., suggested trade-offs between quality of solution and time allocated to processing. This is less immediately applicable to evidence marshalling, which is an all-or-none problem, but much more readily to domains like UAV mission planning, which has both clear quantifiable quality metrics for the solution and often more pressing time constraints, where this type of cost-accuracy trade-offs are much more prevalent. The current analysis in those domains would be framed more in terms of weighing expected improvement in solution quality vs. expected effort allocation, but the analysis would remain fundamentally the same.

Note that these options are not necessarily exclusive. The system could decide to allocate all its resources to the best option (winner take all), it could decide to allocate some resources to each option by an amount proportional to their expected gain (balancing exploration and exploitation) or anything in between, e.g., allocate resources to one or more options with a probability matching their expected gain (probability matching). This depends partly on whether the hardware level naturally allows multiple concurrent processes at low configuration costs, and again trade-off constraints potentially passed from the application layer in terms of risk allowed for quality of solution. For instance, one choice could offer a fairly certain but small increase in solution quality vs. another that would offer a much more sizable but uncertain improvement in solution quality. That choice is ultimately a domain-specific strategic decision best left to the application layer. One could imagine embedding it into the cognitive layer itself as a meta-cognition analysis, primarily at the Macro level, but such an approach would be largely complementary to this one and will be left aside here.

There is already an issue of grain scale of allocation. This decision cycle could be applied at very different levels of temporal and resource granularity. The optimal level at which it should be applied will vary depending on factors such as efficiency of resource allocation algorithm, hardware cost of shifting resource allocation, and such.

As mentioned in the introduction, the above discussion only specifies the information entering into the decision-making process and leaves open the question of how best to implement it. The fundamental constraint derived from this analysis is that no cognitive level (Proto, Micro or Macro) is able to make the allocation single-handedly because the best decision is ultimately a matter of weighing costs and benefits offered by all levels before deciding which one(s) gets more processing allocation. The nature of the control process is ultimately of collecting information from each level to build predictive statistics on which basis to make future decisions, then reconciling them to allocate a common resource. As such, one could view the algorithm as decentralized (e.g., if each level keeps its own statistics and submits them to a central arbiter, e.g., a market-type mechanism, with its requests for more processing) or centralized (e.g., if the statistics are kept in a central location and used to allocate more power when each level needs it) but it is largely irrelevant to the information flow leading to the decision.

One could argue that the latter is ultimately more natural because many if not most statistics ultimately depend on multiple levels of the cognitive system, e.g., the value of Proto improving its clustering coherency ultimately depends upon how much it improves the efficiency by which Micro or Macro reach a decision, or the probability of Micro or Macro finding the right inference (i.e., which one to allocate processing to) might depend upon the quality of the clustering. This kind of system would work as follows:

1. Each cognitive level publishes through cog-CML messages addressed to the controller (as such invisible to the other cog components and thus not burdening them) statistics of their operations including quality of the solution as a function of processing step (e.g., clustering coherency per cycle for Proto, inference quality per instantiation for Micro, etc).
2. The controller gets additional information about the operations of the system through monitoring of the CML traffic (i.e., when a successful inference is reached and by whom—e.g., if both Micro and Macro were allocated processing it is essential to know who first notifies CML of the inference) and the hardware layer (i.e., how many computational resources were consumed by each layer for each allocation, e.g., how long Micro took to instantiate the next inference).
3. Each cognitive level issues a request to the controller for more processing resources when its current allocation (e.g., in terms of time or number of processing steps) is exhausted and waits for the decision.
4. The controller takes into account the information accumulated above as well as the cost-accuracy constraints specified through app-CML to decide to which cognitive level(s) to allocate more processing resources. The cycle repeats.

This system can be viewed as centralized because of the existence of a controller outside of the cognitive levels making decisions about resource allocation or decentralized because the process is driven by the cognitive levels publishing information about their operations and issuing requests for further processing (or not).

3.4.4.2 Algorithm

As described in the previous section, a key requirement of the algorithm is that it has to be adaptive. This proposal takes that into account that by always evaluating each cognition in terms

of its past performance and using those statistics to compute future allocations. Nothing is pre-set: better performance yields more future allocations, poorer performance will decrease them. The requests from the cognitions cannot be assessed upon a constant, unchanging scale (e.g., the urgency and base rate calculations) but instead need to be elaborated to reflect marginal utility as inferred from past performance of the system as a whole.

The basic conception of the process is a sequence of consecutive actions, e.g., Proto-Micro-Micro-Proto-Micro-Micro-Macro (each referring to activities by the respective cognitions, i.e., inferencing and reasoning), followed by an improvement in solution (either binary, as in SoC, or quantitative, as in Unmanned Air Vehicle (UAV) Mission Planning (UMP)) or a failure. This process is repeated for a number of sub-problems (i.e., inferences in SoC and local optimizations in UMP) until the complete problem solved (i.e., the full Wigmorean tree constructed in Soc and the full mission planned in UMP). The controller's basic role is to learn which sequences of actions tend to lead most efficiently to successes in past sub-problems, and use that to schedule those actions in future sub-problems. This is a machine learning problem, and one class of algorithms well suited to this kind of problem is reinforcement learning.

Reinforcement learning [Mitchell97] is framed in terms of a system composed of a set of states S and possible actions A. Each step involves an action transitioning the system to a new state accompanied by a possible reward r. Reinforcement learning algorithms learn the optimal policy π for selecting the action a for each possible state s that maximizes future rewards, with each reward being discounted at each step by a cumulative factor γ.

This definition matches our control problem well. The set of possible actions A from the point of view of the controller is the set of cognitions competing for computational resources, i.e., Proto, Micro and Macro. The rewards will vary for each domain but can usually be defined fairly straightforwardly, e.g., a reward can be associated for each successful inference in SoC (and perhaps a larger one for reaching an actionable conclusion) and rewards can be mapped directly to quantitative solution improvements in UMP. The set of states S for each problem is potentially somewhat trickier to define. Generally, it seems that one could define each state in terms of the amount of processing expended by each cognition on this sub-problem. For instance, in SoC, one could define a state as 1000-2-0, meaning that Proto has expended 1000 cycles on clustering, Micro has attempted to match two inference instances, and Macro has spent no (0) cycles on this problem yet (not clear what Macro cycles consist of). The reason for defining states in this way lies in the Markovian assumption of reinforcement learning, namely that future success is only determined by the current state and future action(s). It would seem that past effort fundamentally defines the probability of future success in our system, and thus should underlie the definition of the current state.

Once the states, actions and rewards have been defined, reinforcement learning algorithms such as Q-learning and temporal difference learning (a.k.a. TD(λ)) can be applied to optimize the system. Basically, the quantity learned is an evaluation function $Q(s,a)$ that associates to every possible state and action pair the expected reward from performing action a in state s. An optimal control system $\pi(s)$ is then merely a matter of choosing in a given state s the action a that maximizes $Q(s,a)$. Theoretically, reinforcement learning algorithms can be guaranteed to

converge to the optimal policy given some assumptions, but in practice no such convergence is guaranteed and a number of parameter choices and techniques have been devised to improve speed and probability of convergence.

A number of additional complexities can be introduced. The system itself should be viewed as non-deterministic, not only because the cognitions and problem themselves are non-deterministic, but more fundamentally because our state representation only captures some of the information that would be necessary to ensure deterministic predictions. Fortunately, extensions of the basic algorithms have been developed to deal with non-deterministic reward and transition functions, which basically consist of averaging successive approximations to the evaluation function as a function of the amount of experience. Conversely, in a non-deterministic system one might want to balance exploration (maximizing results now) and exploitation (learning more about the system and perhaps discovering a better solution) by selecting the next action in a probabilistic manner weighted toward actions with the best Q value(s) rather than always selecting the best one. Again methods have been developed to achieve that goal.

Another potential complexity is that the Markov process assumes that only one action is taken at any one time. However, if time is critical and sufficient computational resources are available, the control system might want to take multiple actions simultaneously, e.g., have both Micro and Macro work on the problem at the same time, or have Micro continuing attempting inferences while Proto improves its clustering. The current algorithm could be extended to that effect by selecting the best n actions rather than the single best one (possibly probabilistically as described in the previous paragraph). Outcomes of that choice could be integrated into the reinforcement learning feedback either by considering each action as a separate branch (e.g., rewarding the one that led to success and punishing the one that did not) or considering as possible actions not only the three separate cognition but also all possible combinations (which only adds another four options to the set of possible actions).

One final but important complexity to consider is that the set of possible states and perhaps actions is really quite large and potentially infinite. If the amount of past effort expended by a cognition spans a wide range (e.g., up to hundred of thousands of Proto cycles, up to hundred or thousand of Micro inferences, potentially many Macro decision cycles), it becomes intractable to represent $Q(s,a)$ as a table and interpolation schemes that generalize from past experience to similar but not identical situations need to be considered. Techniques to accomplish that goal include representing the evaluation function Q as a neural network, e.g., trained by back-propagation (such as done in Tesauro's world-champion backgammon program). A variation that has been showed to be quite successful in practice is to train one network for each possible action, taking as input the current state and outputting the Q value of this action applied to this state. This could be an intriguing for our application because it would allow the controller to change from a central algorithm to one distributed over each cognition in the form of a neural network leaning the Q values for that cognition. Similar techniques could be used if a fixed grain scale for allocating processing resources is not suitable but instead the controller must also decide how much additional resource to allocate given a selected action (e.g., how many more Proto cycles?).

3.5 Design of the Hardware Layer

3.5.1 General Architecture

The hardware design includes support for common tasks required by the cognitive engines, and provides hooks for other PCAA concepts such as dynamic scheduling, resource management, and power management. Power management is extremely important because reducing the power consumption results in a smaller form-factor: chips are small, power supplies and cooling systems are not.

The hardware architecture is based upon tightly-coupled processor, high speed configurable logic, specializable memory, and communication circuitry. The configurable logic is based upon dual-line asynchronous signaling and can implement mission-specific designs at near-custom ASIC speeds.

A high-level view of the overall hardware architecture is shown in Figure 22. The main components of the architecture include:

- Partitioned memory, with an off-chip memory interface and a dynamically adjustable section for fuzzy associative memory in each tile.
- A fast, low-complexity processor that is highly power efficient (e.g., 2-way or 4-way VLIW) and that has hooks for monitoring and voltage/frequency scaling.
- A high-speed on-chip network and a seamless corresponding chip-to-chip interface.
- Local and global clustering support, for larger "virtual tiles."
- Fine-grained reconfigurable logic (Asynchronous Field Programmable Gate Array (AFPGA)) that can also be clustered.

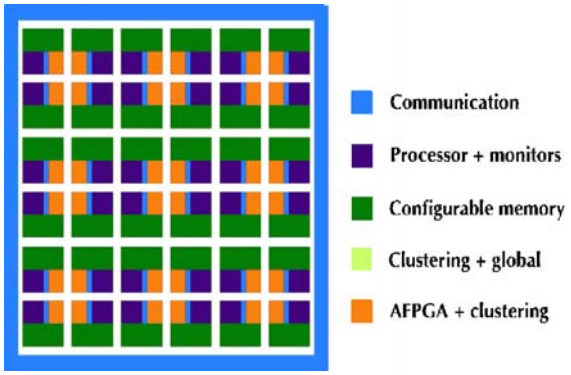


Figure 22. CHASM topological design

The rationale behind the building blocks is explained in detail below. The key computation elements support fine-grained processing using an asynchronous Field Programmable Gate Array (FPGA), and coarse-grained and adaptive processing using a tile-based array of processors. The memory system is partitioned, and can be dynamically configured to support normal memory access as well as associative operations.

Figure 23 shows the organization of an individual tile. Each tile contains an array of memory banks that are immersed in a programmable and pipelined switch-point based network. This network is similar to the high-performance routing network in asynchronous FPGAs [Manohar04]. The memory can be configured to communicate directly with the processor via the processor-memory interface, or the AFPGA via the AFPGA-memory interface, or both. The large number of individual banks can be configured as a single memory by grouping the address decoding and data transfer, or as a partitioned set of memory blocks that are individually addressed.

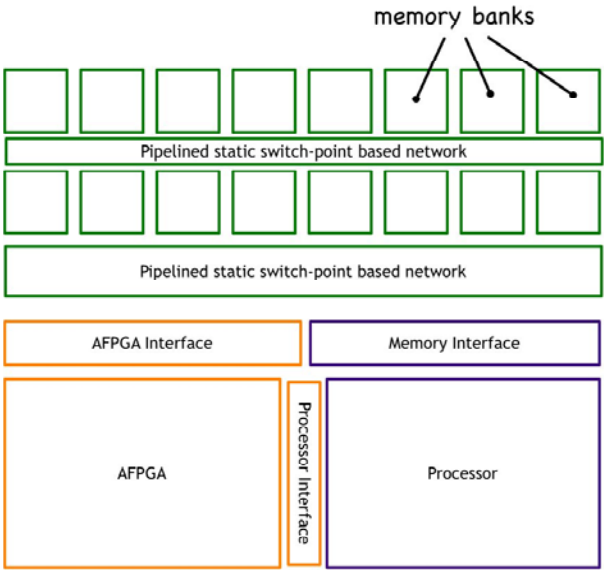


Figure 23. Layout of an individual tile

The AFGA can communicate directly with the processor through an interface that is connected to the processor bus. Thus, data transfers can be setup from the memory to the processor that include AFPGA post-processing or pre-processing.Memory-to-memory transfers that involve dedicated AFPGA logic for processing are also possible in this architecture.

3.5.1.1 Clustering for Dynamic Resource Management

Support for dynamic resource management is provided via clustering support. Groups of hardware tiles can be tightly coupled to form large clusters. These clusters can cross chip boundaries, at the expense of higher intra-cluster communication latencies. Top-down application-driven clustering is controlled via.the processors in each tile. The net effect of clustering is to form multiple "virtual processors" from the physical resources. Monitoring is provided to support dynamic and feedback-directed clustering.

Another mode of resource sharing is to combine different aspects of each tile. For instance, we might combine the asynchronous FPGAs (AFPGA) from four tiles to form one larger virtual AFPGA, while also grouping their processors together. The AFPGA can execute Proto cognition tasks, while the processors can execute Micro and Macro-cognitive tasks. This support is provided by the inter-cell/inter-tile network and routing (Figure 24).

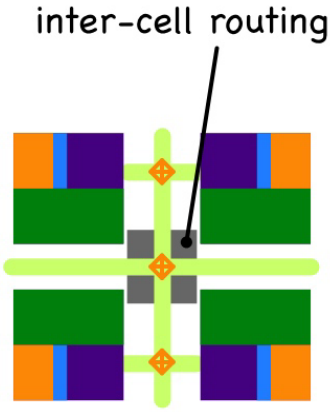
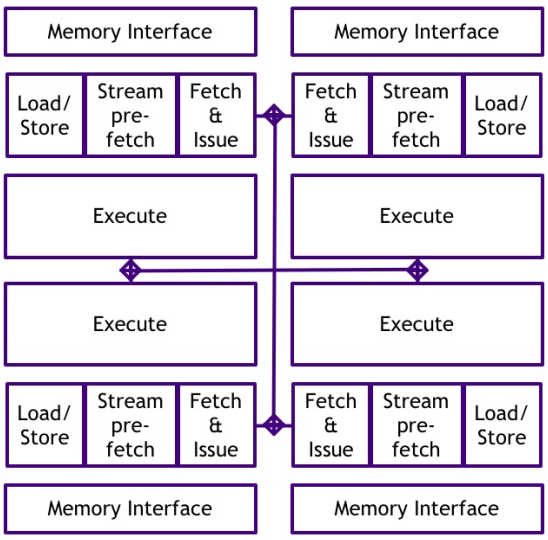


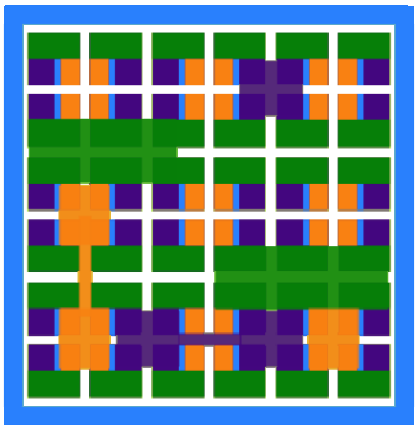
Figure 24. The inter-cell/ inter network and routing

3.5.1.2 Multi-granularity Parallel Execution

Multiple granularities of parallel execution are provided in two ways. First, fine-grained processing is supposed by the AFPGA architecture. This can be configured like a traditional FPGA to provide support for very fine-grained processing and bit-level operations. Secondly, a partitionable processor that can be configured as either a wide-issue or a single-issue processor, depending on the needs of the application, provides coarse-grain adaptivity.

The processor consists of the core datapath (labeled "Execute") and memory access blocks. The memory access blocks include the load/store unit, a programmable pre-fetch engine, and an instruction fetch-and-issue block (Figure 25). The fetch-and-issue blocks of four processors can be combined using a programmable interconnect to either operate independently or in unison. This can be efficiently implemented in a VLIW architecture because the fetch-issue operation is





Example of clustering, with different tiles merged into larger chunks based on needs of different cognitive tasks. The clustering can change ***dynamically***.

Figure 25. A programmable pre-fetch engine and an instruction fetch-and-issue block

naturally partitionable. Additional programmable interconnect is required among the execution units so that register references can be implemented as either local access (if the processors are partitioned), or transferred between execution units (if the processors are grouped into a single processor). This permits the amount of concurrency to be varied based on the amount of available instruction-level versus thread-level parallelism in the application.

3.5.1.3 Associative Matching and Memory Block Transfers

Configuring the memory and FPGA blocks to communicate directly with each other supports associative matching. The matching operation that we demonstrated using an FPGA-based Prototype can be implemented using this configuration.

As cognitive engines like ACT-R and SOAR can predict what their future memory access might be, we provide support for a flexible stream-based pre-fetch engine in each processor. This engine is used to schedule data transfers from off-chip (or remote memory) in a predictive as well as application-controlled manner. Stream pre-fetchers are highly effective at hiding off-chip memory access latency. However, software-controlled pre-fetch operations will also be supported so that we can improve performance when it is known what the future memory access pattern will be.

3.5.1.4 System Monitors

Local hardware monitors in each tile will provide the necessary hooks for system measurements. These measurements can be used to dynamically tune execution (e.g., determine if more/fewer resources are needed to execute a task). In order to allow for efficient power management, a number of hardware monitors will be integrated in the architecture. Examples of these monitors are access counters and time stamps for memory blocks. Hardware monitoring is important for

cognitive applications. The hardware must provide feedback to the cognitive layer about the properties of the current computational stream. Traditional applications act based on their current state and new inputs. Cognitive applications base their actions also on the current availability of hardware resources.

We envision four kinds of hardware monitors: extended event counters, real-time timestamps, quality metrics for chip-resident cognitive features, and history buffers for control and data information.

3.5.1.4.1 Extended Event Counters

The PCAA architecture will extend event counters present on modern architectures with more general metadata about the computation and the hardware resources it is consuming. For instance, in addition to a counter for the number of data cache misses, the PCAA architecture will have a counter for the rate of change of the number of data cache misses (its discrete derivative with respect to time). Data cache misses will accelerate when a program changes phase from working on one set of data structures to working on another. By reading these counters and using machine learning, the cognitive virtual machine can model its own behavior in order to allocate its resources most efficiently, and to predict its future resource use.

3.5.1.4.2 Timestamps and Timestamp Storage Buffers

Another form of metadata in the PCAA architecture is realtime timestamps. By providing occasional realtime timestamps for chip events, the PCAA architecture enables software to correlate activity in different parts of the chip. The PCAA architecture will have a timestamp for each page of DRAM. If a particular page is accessed soon after the receipt of a clock interrupt, and only soon after the receipt of a clock interrupt, it indicates the data structure accessed by the clock interrupt handler is on a cold memory page. The system can prefetch the memory, or move the data structure to a page which is usually hot when the clock interrupt arrives. This use of hardware counters can improve performance and reduce energy consumption.

The hardware will only record realtime timestamps occasionally, not every time an event occurs. Frequent events, like branches, would require too much circuitry to timestamp each instance and too much memory to store all those timestamps. How to produce timestamps that are useful to the software monitoring layer is a key research challenge. Software might identify a set of resources whose use should be timestamped at the same sampling rate into a circular memory buffer.

3.5.1.4.3 On-chip Cognitive Metrics

The PCAA architecture will provide feedback about the use and quality of results of the cognitive features of the chip. For instance, the accuracy of an approximate match from the content addressable memory. The software can detect low utilization of hardware and adjust its problem solving strategy, or instance moving data from a small associative memory to a larger, slower hash table.

3.5.1.4.4 Control and Data History.

Cognitive applications will examine their own execution history in order to increase efficiency and increase functionality. Execution monitoring can increase efficiency if, for example, the runtime system notices periodic behavior, it can prefetch data; it can reschedule computation to make greater use of execution resources.

3.5.1.5 Asynchronous FPGA

Fine-grained concurrency and the static flexibility of the architecture are provided using a high-performance reconfigurable fabric implemented with asynchronous logic. The use of asynchronous logic in an FPGA leads to a number of benefits, the primary one being performance. By developing a highly pipelined architecture, we have demonstrated an FPGA that can operate at a frequency that is 16-18 fanout of four delays per cycle. In 90nm CMOS technology this translates to a performance in the 1.5 GHz range, and corresponds to over 2 GHz performance in 65nm CMOS.

The core of the FPGA is implemented as a reconfigurable static dataflow machine, and computations are mapped to the architecture by compiling them using conventional synthesis techniques. While we use novel circuits and a new micro-architecture for the FPGA, at the architectural level the FPGA resembles a standard, island-style FPGA with an array of logic blocks and switch boxes for the interconnect (Figure 26). This permits conventional place-and-route algorithms and logic mapping techniques to be used for mapping designs to the architecture.

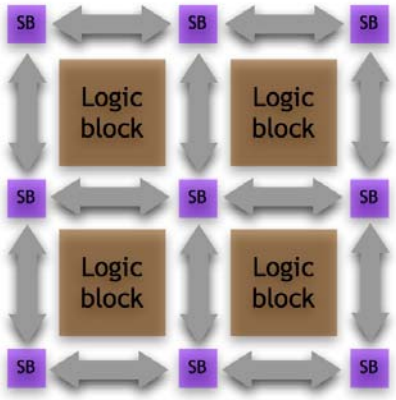


Figure 26. The FPGA resembles a standard, island-style FPGA with an array of logic blocks and switch boxes for the interconnect

The tight coupling of the asynchronous logic with memory and processors allows us to implement programmable inline address and data transformations needed for structures like POEM. For a full discussion of the asynchronous programmable logic to be used for this design, see [Manohar06] and [Teifel04].

3.5.2 Hardware Devices for Cognitive Components

Experimentation with Macro Cognition based on SOAR [Newell00], Micro Cognition based on ACT-R [Anderson98], and Proto Cognition based on swarming [Parunak04] have guided implementation studies of the hardware. SOAR applies production systems in service of explicit goals; in each execution cycle, all productions whose precondition is enabled by the content of memory will fire, potentially updating the working memory or rule base. SOAR has designed-in mechanisms for resolving conflicts and integrating knowledge from multiple sources. ACT-R emphasizes reactive problem solving through expertise-based pattern matching. In our system, ACT-R also provides the interface between the Macro cognitive symbolic operations of SOAR, and the Proto cognitive sub-symbolic operations provided by Swarming computation. Swarming

can implement perception processing effectively, using hierarchical ant clustering, extracting cognitive elements from massive amounts data.

There is an apparent correspondence between our chosen cognitive frameworks and the UAV mission planning problem as framed algorithmically that arguably should provide problem simplification. For example, swarming is a well-known meta-heuristic used in the solution of graph problems. The chunking operation in SOAR is an Explanation Based Learning procedure that is analogous to the conflict clause generation procedure found in fast constraint solvers. The caching and approximate matching in ACT-R can leverage previous knowledge of good solutions that simplify the generation of new plans. But we are more ambitious in our desire to apply the cognitive frameworks; for example, we would like to investigate having SOAR reasoning about the problem in terms of an upper ontology for very high level problem simplification.

Our architecture is also an experiment into the fusion of the different cognitive models into one intelligence; we have prototyped a cognitive markup language (mCML) to implement communication between the intelligences. There are numerous opportunities for further investigation of such a configuration, such as finding ways to use the heuristics power of swarming or ACT-R to make the reasoning in SOAR tractable. Furthermore, we would like to investigate integrating other types of cognitive models into the fusion, such as for probabilistic reasoning to model uncertainty or subsumption architectures for greater integration of the UAV control system.

Another challenge is the degree and manner of integration of the existing algorithmic approaches to mission planning with the cognitive architectures. It is an open question which is better: a fine-grained fusion of such techniques (e.g., implementing the search directly in one of our frameworks) vs. coarse partitioning (coupling a highly tuned constrained A* solver with a highly tuned implementation of a cognitive architecture). An implementation of A* in our cognitive frameworks, for example, is attractive from the point of view of a seamless transition between that path finding and the high level reasoning, but it could be computationally very inefficient.
Hardware Support for Micro Cognition through ACT-R

Micro Cognition, based on ACT-R, employs memory intensive operation for expertise-based solution using three logical memory compartments: the declarative memory is used to symbolically represent compositional data, the procedural memory stores information for reaction in dynamic problem solving, and the working memory maintains the current operational context. Each memory type operates on similar principles for matching. The match operation is inexact in that the data associated with the closest key in memory (according to a definable distance metric) is retrieved. The similarity function, which compares an input operand to each content row (or so-called chunk) of memory may be further compounded by factors such as decay or practice.

Our hardware designs for ACT-R leveraged the inherent parallelism in the independent comparisons of chunks with input operands.

3.5.2.1 Coarse-Grained Processor Parallelization for ACT-R

Experimentation was performed on parallelizing the kernel computations of ACT-R on general purpose processors. The experiment focused on declarative and procedural memory since they account for most of the computation time in ACT-R. Figure 27 shows the computation graph for the core ACT-R operation examined.

The implementation was done on power PCs using MPI. The experiment also has implications to load balancing strategies and power management since the number of rules and chunks changes over time. Figure 28 shows the results of speedup vs. number of Power PCs cooperating on the match operation.

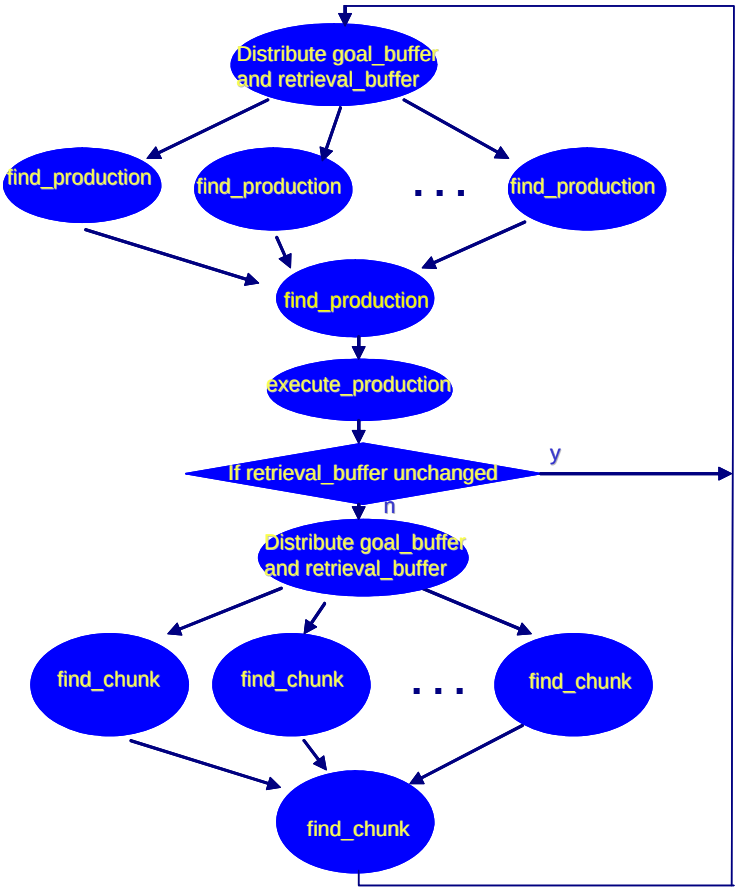


Figure 27. ACT-R Computation Graph

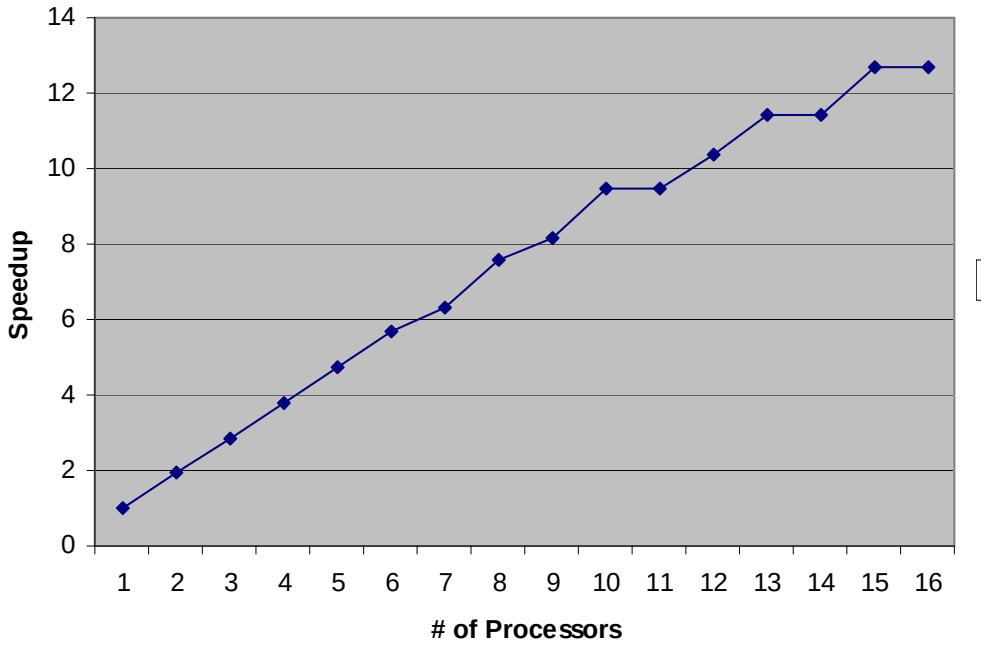


Figure 28. Speedup of ACT-R vs number of processors

3.5.2.1.2 POEM Specialized Memory Acceleration for ACT-R and Memorization

Further speedup (up to two orders of magnitude) was achieved by using a custom inexact matching architecture (implemented in a Xilinx Virtex-2 FPGA) shown in Figure 29.

The programmable objective evaluation memory (POEM) of Figure 29 implements a user-programmable match function (F) in parallel across 256 bits of the input operand, along with parallel banking of memory, objective minimization, and result selection. The Virtex-2 implementation can sustain in excess of a 50 mega-chunks-per-second match rate for a fully pipelined match function, where a chunk in this case is a vector of 8 32-bit words. The "C" code portion used to synthesize the design is shown in Appendix Section 7.3.1. VHSIC Hardware Description Language (VHDL) code is too lengthy to include in this report but can be made available to authorized parties.

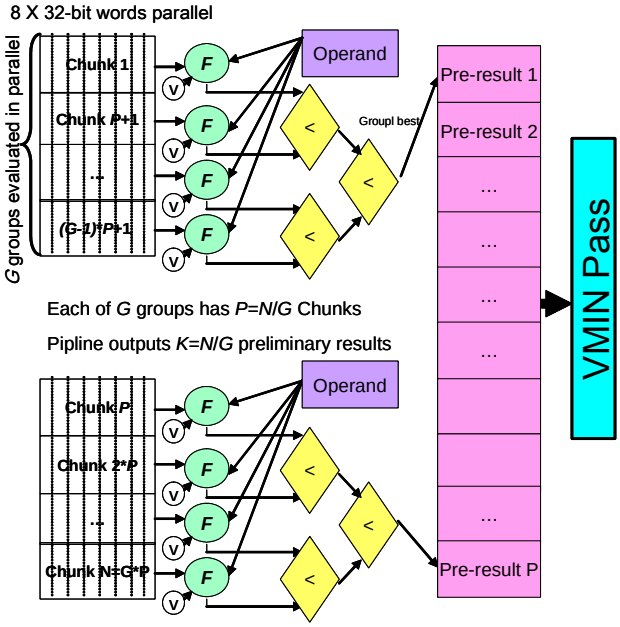


Figure 29. Matching Memory for ACT-R acceleration

This memory can perform exact and inexact matching depending on the choice of "F." In exact matching mode, the memory supports hardware level memorization (or functional caching). If the function is set to multiply and accumulate (MAC) and bias, the memory implements a parallel perceptron selection bank and extends to support vector machine functionality with an additional transform implemented in "F." For our cognitive architecture, the main use of this specialized memory is for ACT-R chunk-matching and for hardware memorization and hash acceleration. In order to evaluate the performance in such chunk-matching, we investigated a simple Traveling Salesman Problem (TSP) mini-application. The Traveling Salesman Problem (TSP) was chosen to test the recall ability and accuracy of such a memory. In learning mode, candidate problems are solved optimally, and the problem/solution pairs are stored. A replacement policy which favors replacing least frequently used content with new inadequately represented content effectively flattens the response error. The code for training the memories is shown in Appendix Section 7.3.2.

3.5.2.1.3 CA-RAM Specialized Memory Acceleration for ACT-R and Memorization

The flexibility of reprogrammable logic allows us to compose memory architectures to suit the application kernels. A second type of specialized memory explored for ACT-R acceleration requires less of the reprogrammable resources, and with some special hash logic, can perform nearly as well as the POEM architecture if minor loss can be tolerated in the matching accuracy. The CA-RAM architecture, shown in Figure 30, separates memory cells from match logic and

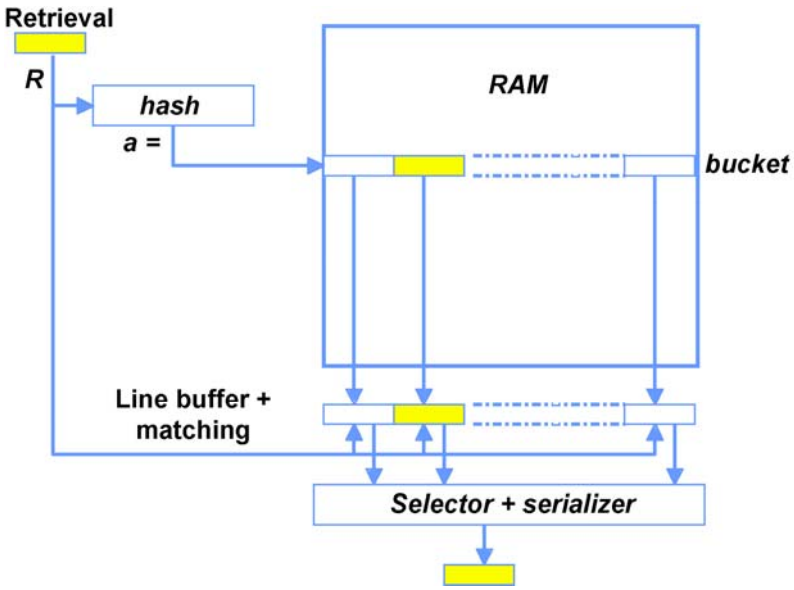


Figure 30. The CA-RAM Architecture

shares this logic among RAM lines, resulting in both large capacity and fast search. The hash function is lossy, and similar to POEM, the match logic selects the entry with the largest score. For example, in the shortest path algorithm, the lossy hash may simply ignore the lower order bits of the coordinates.

3.5.2.2 Hardware Support for Macro Cognition through SOAR

Macro Cognition is based on SOAR. The core production matching of SOAR targeted for hardware implementation is based on an algorithm known as Rete [Forgy82]. Rete is an exact-match algorithm that uses two memory types: the *working memory* contains facts about the world that are collected over time, and the *production memory* contains rules. Each rule in the production memory would typically be a set of conditions, and a set of actions to perform if those conditions are met. An added complexity is that the condition expressions may contain variables which must be bound during matching.

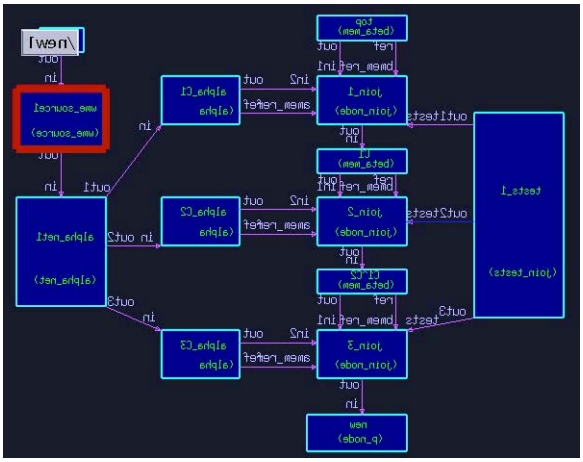


Figure 31. Rete in event-driven logic

The study of Rete hardware targets a simulation of the event driven processing afforded by asynchronous logic [Manohar04]. Figure 31 shows an asynchronous data flow network for a Rete, where each node is built from fine-grained logic, and can execute concurrently as soon as dependencies are met. As is typical with asynchronous approaches, energy consumption for computation is reduced by computing only when necessary, and performance can be greater by removing the need for a central clock.

Due to the dependencies of typical networks, the maximum concurrency achievable is only on the order of 10X to 20X. Hardware support for hash functions yields higher constant factors of acceleration and are under consideration. The library used to exercise the Rete experiment is shown in Appendix Section 7.3.3.

3.5.2.3 System Support for Proto Cognition through Swarming

To support the Proto Cognition, a key cognitive component of the PCAA architecture, we attempted to abstract the common control aspects of its swarming algorithms through development of an API. The first level of this abstraction, ACIPL (Advanced Cognitive Information Processing Library), shown in Appendix Section 7.2.1, isolates the developer from the configurable logic through use of optimized domain-specific generators. The second level of abstraction, AVML (Agent Virtual Machine Language), shown in Appendix Section 7.1.3, allows access to low-level parallelization primitives. Whereas swarming algorithms have traditionally been written as custom applications, often from scratch, the API abstraction is useful to allow more sophisticated and efficient use of the underlying hardware. By handling low-level details, this reusable infrastructure also naturally helps support rapid cognitive application development.

Swarming is conceptually highly parallel. In our API Prototyping, we found this parallelism to be readily available in practice. The challenge is often the grain size, however; with too little work per node, communication costs can dominate. We have found it will be important to manage the grain size issue through clustering of highly-connected nodes and communication-aware layout of work, including dynamic rebalancing.

3.6 Metrics for Evaluating a Cognitive Architecture

In addition to developing the C3I1 architecture, as discussed above, we also need to evaluate it. The problem of evaluating general architectures is a difficult issue [Newell90] because it is usually feasible to develop a specialized solution for any particular problem that will outperform the general solution. Thus, in general, the evaluation of a general architectural approach derives from the power of the primitives of the architecture, the generality and flexibility of these primitives in providing solutions across a range of tasks, and the resulting ability to (relatively) rapidly develop an architectural solution via a common computing framework.

The following section introduces a number of metrics that could be important for assessing the PCAA cognitive layer. While efficient performance is a necessary ingredient for effective behavior, it is by no means sufficient. These metrics together reflect our attempt to capture and measure *all* necessary components of general intelligent behavior. The taxonomy is based on an analysis of human cognition by Anderson & Lebiere [Anderson03]. We focus only on the functional aspects of their analysis, rather than those non-functional requirements specific to human cognition.

One of the primary factors that makes achieving general, intelligent behavior such a difficult problem is the complexity and variation found in the environment. For general intelligence,

agents must be able to cope effectively with this complexity. However, while complex, the environment (usually) is not chaotic. It operates according to laws and general properties and can be characterized according to its complexity. Table 3 illustrates one characterization of environmental complexity, adapted from an AI textbook [Russell95]. Different problems will have different complexity profiles based on these dimensions. Many of the metrics introduced below will also interact with these dimensions, so that quality of the overall solution and the problem complexity define a functional space for the metric for some particular solution.

Table 3. Dimensions Characterizing AI Problem Domains [Russell95]

Fully observable vs. partially observable:	**If an agent's sensory apparatus gives it access to the complete state of the environment then we say that the environment is fully observable. An environment is effectively fully observable if its sensors detect all aspects that are relevant to the choice of action. A fully observable environment is convenient because the agent need not maintain any internal state to keep track of the world.**
Deterministic vs. stochastic:	**If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say it is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the environment is partially observable, however, than it may appear to be stochastic. This is particularly true if the environment is complex, making it hard to keep track of all the unobserved aspects. Thus, it is often better to think of an environment as deterministic or stochastic from the point of view of the agent.**
Episodic vs. sequential:	**In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends on just the episode itself, because subsequent episodes do not depend on what actions occur in the previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.**
Static vs. dynamic:	**If the environment can change while the agent is deliberating, then we say the environment is dynamic for the agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is semi-dynamic.**
Discrete vs. continuous:	**This focuses on the way time is handled and to the percepts and actions of the agent. If there are a limited number of distinct, clearly defined percepts and actions we say the environment is discrete. Chess is discrete - there are a finite number of states and a discrete number of percepts and actions. Taxi driving is continuous.**

3.6.1 A Taxonomy of Evaluation Criteria and Associated Metrics

We identified nine general categories of evaluation criteria for the C3I1 architecture and then proposed dimensions of evaluation within each category. In most cases, we also identify specific metrics for these dimensions. However, in some cases, qualitative and/or subjective evaluation may be necessary.

3.6.1.1 Behave as an (Almost) Arbitrary Function of the Environment Exhibit Computational Universality

The environment can change independently of the intelligent system and the actual state of an environment may not be known. Thus, the intelligent system must be able to act in the situation it finds itself in (and even if it is different than the one it expected to be in). Flexibility implies a breadth of capability, meeting the complexity of the environment with appropriate responses.

3.6.1.1.1 Taskability

Taskability is the ability of a system to adapt to new/novel problems without human (programmer) intervention. Taskability is difficult to measure because there is no "absolute"— a measure of "50" for one domain might represent the best one could achieve, while in another, it might be a baseline. Researchers in AI have generally evaluated taskability by adopting a set of benchmark tasks against which a system is developed, and then introduced novel tasks within the same domain and tested system performance on these new tasks ([Hanks93]). This approach provides a reasonable qualitative measure of taskability within a domain.

3.6.1.1.2 Incrementality

Incrementality is the ability to extend the system from one set of tasks to another set, which can be either a superset of the original set (generalization) or somewhat different set of requirements (robustness/taskability). Incrementality could possibly be measured by the degree of overlap between the solutions to the two sets of problems. For instance, if the cognitive systems provided are quite general, a small task-specific addition at the application level might be sufficient to tackle a new task. On the other hand, if they are overly specific, then significant reworking for the new task might be necessary and the resulting incrementality will be poor. Evaluating incrementality will expose excessive benchmark-driven task specialization and thus helps ensure the generality of the architectural framework. To our knowledge, incrementality is a novel dimension that has not been previously evaluated.

Figure 32 illustrates a notional approach to evaluating incrementality. A very simple measure of incrementality could be the ration of unchanged lines of code to total lines of code for a particular problem, in comparison to previous problems. In the ideal case, a long-term, not near-term goal, full incrementality is reached, in which no new changes are needed for a new task. In contrast, in the current state-of-art, there is little reuse across problems, resulting in very shallow incrementality. The goal of a system like C3I1 in PCAA is to significantly increase reuse across problem domains, as shown in the solid, black line. While full incrementality is likely infeasible, this approach would provide at least a coarse measure of incrementality for cognitive systems and allow us to assess the level of improvement in this dimension.

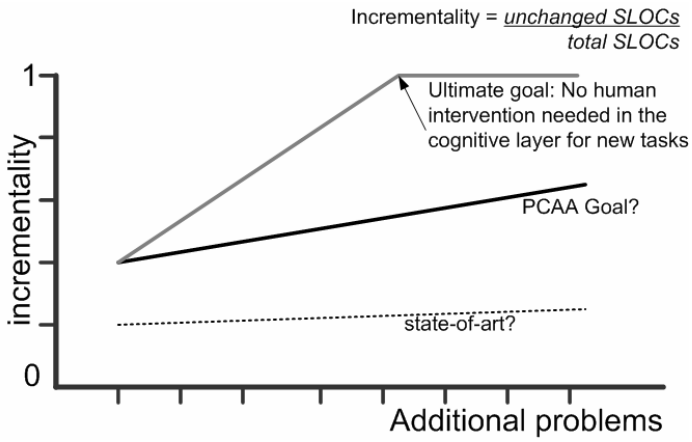


Figure 32. A notional approach to evaluating incrementality.

3.6.1.1.3 Operate in Real Time

Does the system respond to the environment and task in a timely fashion? For an intelligent system to behave intelligently, it must able to recognize a situation that it cares about, determine an appropriate response, and then act. However, the world in which an agent operates may be continually changing. Thus, the agent must perform its internal processes quickly, relative to the speed of change in the environment, or its chances of survival/success diminish in a ling-term-existence. Many reflexes and instincts can be viewed as evolutionary solutions to the problem of fast reaction in the animal world, "hard wiring" responses to specific situations. The important requirement is that the speed of response is sufficient for the demands of the problem, rather than being as fast (in absolute terms) as possible.

3.6.1.1.4 Response Time

Response time is the time between the onset/assignment of a task and its resolution. Response time should have a Quality of Solution associated metric, to distinguish between satisficing/non-satisficing solutions, and to demonstrate trade offs between solution quality and response time. Also, response time should distinguish between soft and hard real time responses.

3.6.1.1.5 Cognitive Cycles/Cognitive Operations

It measures the cycles (or percentage of total CPU time) devoted to cognitive operations. This metric is a poor stand-alone metric because decreases or increases cannot be evaluated in an absolute sense, similar to the way comparing operations per second in RISC vs. CISC architectures is also marginally informative. However, cognitive operations/time is a good *relative* metric, allowing one to assess improvements against a baseline or benchmark.

3.6.1.1.6 Scalability

How many cognitive services/"cognitive operations" can be supported per some fixed unit of processing power.

3.6.1.1.7 Extended Operation/Longevity

What is the uptime of the cognitive system? How do other performance metrics change as uptime increases? For example, does system performance degrade as uptime increases (this is often observed in some learning systems, where the addition of knowledge via learning leads to significant degradation in knowledge retrieval performance)?

3.6.1.2 Exhibit Rational, i.e., Effective Adaptive Behavior

Does the system yield functional behavior in the real world? An intelligent system must not only respond to its environment, but it should respond in a manner appropriate for the situation. In particular, as the world changes, the agent should adapt its behavior to the situation such that it continues to make progress on its long-term goals (i.e., it cannot just be reactive). Because

environments have consistent (or slowly changing) dynamics, an agent can make predictions about future states and attempt to act to effect the environment in ways that meet its goals. Different sources of knowledge available in the environment can be used to formulate goals, to act to achieve them, and to recognize when goals are met or unreachable. Adaptation to the specific environment is important because the agent's existence may span a long period of time, and non-adaptive behavior may influence the survivability or viability of the agent in an application domain.

Specific metrics include dimensions introduced previously, but with different emphases, points of comparison, in this category.

3.6.1.2.1 Response Time

What is the response time to an unexpected event? Is the system able to resume execution of an existing plan once an interrupting event to which the agent has responded? As one example, [Wray03] developed a domain specific metric to illustrate the reaction time of a system in response to a triggering event, but there is no domain general metric for evaluating this aspect of response time.

3.6.1.2.2 Scalability

Ability to handle increasingly complex problems along the Table 3 dimensions characterizing cognitive problems. Collective, subjective judgment is currently the only known approach to ranking specific domain problems along each dimension.

3.6.1.3 Use Vast Amounts of Knowledge About the Environment

How does the content and size of the knowledge base affect performance? There are many different objects in the environment, including other agents. Most objects obey consistent, predictable dynamics, although the agent may not have complete or correct knowledge about these laws. These attributes of the environment make "knowledge" a fundamental requirement for intelligent systems. "Knowledge" here really means nothing more than having the means to predict future states of the environment; it does not necessarily imply deep, first-principals knowledge (e.g., the Three Laws of Thermodynamics). However, as the agent's environment becomes more complex (in terms of the objects and interactions it must manage to succeed), it will need increasingly large stores of knowledge to cope with the complexity.

3.6.1.3.1 Size of Knowledge Base

How much "knowledge" is represented in a given cognitive service and the overall cognitive layer. Within a particular symbolic cognitive service, it should be relatively straightforward to characterize the size of a knowledge base. However, this metric is difficult to quantify generally. How do we characterize "knowledge" in non-symbolic systems? Although counting knowledge representations is trivially simple in symbolic systems, this can also be misleading (a SOAR rule

and ACT-R rule correspond to different grain-sizes of cognitive operations; is an Ontology Interface Layer (OIL) sentence comparable to a rule in an cognitive architecture, etc.?)

3.6.1.3.2 Response Time

How does response time change when one-to-two orders of magnitude of "knowledge" are added to the system or individual cognitive services?

3.6.1.3.3 Cognitive Cycles/Cognitive Operations

How does COPS change when one or two orders of magnitude of "knowledge" are added to the system or individual cognitive services?

3.6.1.3.4 Cognitive Footprint

How do hardware memory requirements scale with knowledge representation requirements? We cannot speculate on specifics of these performance issues for future systems but we elaborate in the conclusion section (Section 5), the experiments and the results we archived.

3.6.1.4 Behave Robustly in the Face of Error, the Unexpected, and the Unknown

Can the system produce cognitive agents that successfully inhabit dynamic environments? An agent will always have incomplete or partially incorrect knowledge of the many objects and other agents that appear in its environment. Yet, in order to thrive, it must overcome these limitations and complexities in the environment and behave robustly. The environment itself provides some important aid in this respect. First, the environment usually has structure and abstractions that alleviate the unpredictability of the situation. One may not have experience with the specific SAM site just encountered in the execution of a mission, but previous experience with and knowledge of anti-air defenses makes the situation more predictable and provides suggestions for courses of action that increase the likelihood of survival. Second, the environment provides many sources of knowledge including direct experience, observation of others, instruction, etc.

3.6.1.4.1 Robustness

Robustness is the ability to successfully (autonomously/dynamically/safely) withstand perturbations in expected events and tasks. There are no existing, general metrics for robustness, although domain specific metrics have been developed [Nielsen02]. One possibility would be to cast robustness as the ratio of the degree of success vs. the degree of perturbation. However, both of these measures are likely to be domain and task dependent. Also note that robustness has a direct relationship with the notion of taskability introduced earlier. The primary difference is that taskability focuses on the ability of the system to handle variation in tasks, while robustness primarily focuses on success in environments where the expected dynamics are changing.

3.6.1.5 Integrate Diverse Knowledge

Is the system capable of common examples of intellectual combination? The objects and other agents in the environment result in many different sources of knowledge. To act appropriately, the agent must put all this knowledge together to act appropriately in a situation. For example, in the evidence marshalling task described in Section 4.2.1, the system must integrate knowledge from intelligence reports, draw on past experience, be able to reason deductively as well as by abduction and analogy, use general knowledge (language, ontology, etc.) as well as domain specific knowledge (terror organizations). While the task could possibly be accomplished without multiple sources of knowledge, the assumption is that the introduction of a much larger branching factor in both knowledge search and problem search is offset by the ability to reach a conclusion in just a few steps.

3.6.1.5.1 Versatility

Versatility can be measured by enumerating the goals, methods, and behaviors exhibited by the cognitive components of PCAA.

3.6.1.5.2 Scalability

How "much knowledge" can be represented with a system given some baseline performance requirements? In general, systems should scale up to increasingly knowledge-rich approaches to problems.

3.6.1.6 Behave Autonomously in a Social Environment/Use Language (Natural or Artificial)

Over a long life of continual behavior in the environment, an agent will pursue its own success and act on its own. However, the agent may live in a social environment with other agents and actors. Other agents can complicate ultimate success (competitors), but may also be the source of additional knowledge and cooperation. Other actors require that the system be knowledgeable of them (able to predict actions and evaluate intents) as well as the ability to communicate with the other actors.

3.6.1.6.1 Scalability

How many "cognitive agents" can interact together, given some baseline performance measure (or processing/power constraints)? Ideally, overall performance costs will increase at most linearly with the addition of multiple agents.

3.6.1.7 Exhibit Self-Awareness and a Sense of Self

Is the system able to recognize problems in its processing and take corrective action? Because existence is long-term, an agent will have many opportunities to recognize deficiencies in its knowledge of the environment, and can use the many different sources of knowledge in the

environment to address the deficiencies. “Self-awareness” is the capability to recognize these opportunities to reflect on the state of one’s self and one’s behavior and to improve future action by evaluating the efficacy of actions taken in the current situation. Self-awareness can also include notions of performance monitoring and fault localization within the overall system (i.e., extending beyond the cognitive components of the system).

3.6.1.7.1 Versatility

What are the meta-cognitive architectural processes and system capabilities? This is a wholly subjective accounting of the kinds of processes available in the system for meta-cognitive activity.

3.6.1.8 Learn from its Environment

Can the system produce a breadth of different types of learning and improve its function? If the world is consistent and the agent’s knowledge is incomplete (as will almost always be the case), then an obvious requirement for long-term success in the environment is learning. Learning, which will draw from the many sources of knowledge in the environment, re-shapes behavior. In the ideal case, learning improves outcomes of future experiences in comparison to past, similar ones.

3.6.1.8.1 Adaptivity

Adaptivity corresponds to learning mechanisms in the cognitive architectures, both in term of short-term adaptivity (change what you are doing in this specific situation) and long-term adaptivity (acquire new general knowledge to improve long-term performance). It is straightforward to quantify the performance gain in a particular domain by comparing a given non-adaptive system to an adaptive version of the same system. This measure is somewhat related to robustness but does not reduce to it. Both robustness and adaptivity are necessary: robustness provides acceptable performance in unforeseen situations, adaptivity simplifies *a priori* engineering and provides a more efficient solution-authoring process than one in which complete capability has to be specified by the designer, in addition to allowing the system to adapt on its own to the changing dynamics and task requirements of a domain.

Figure 33 illustrates a possible, notional approach to a general measure of adaptivity. In this approach, adaptivity is the ratio of the size of perturbation in the environment to the difference in the resulting quality of solution. As the difference in quality of solution increases (presumably, a poorer quality solution), adaptivity decreases. Similarly, if quality of solution remains constant while the perturbation increases, adaptivity also increases. In current state-of-the-art, systems are generally designed for a given problem complexity and quality of solution, so that any perturbations, even ones in which complexity decreases, quality of solution is likely to decrease (with very sharp decreases as complexity increases). The middle line in the figure illustrates a possible goal for adaptivity in which quality of solution decreases more gracefully as problem complexity increases. Further, for a particular point on the problem complexity curve (stars), learning should result in an improved quality of service (improved performance or solution

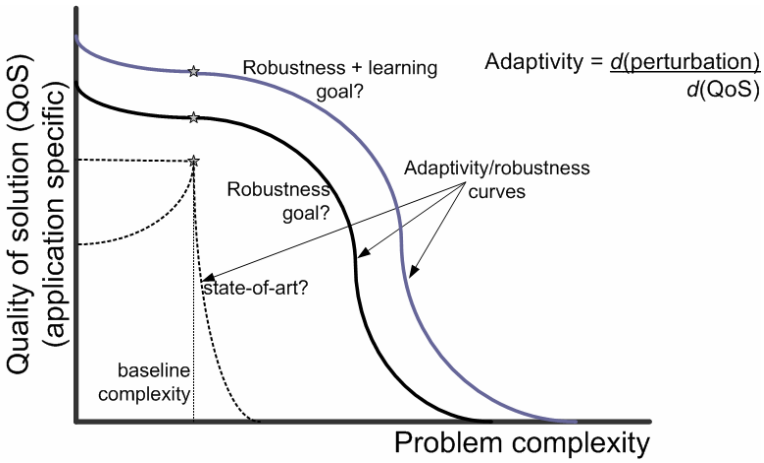


Figure 33. Notional proposal for adaptivity measure.

quality). Thus, one view of learning is that it allows the adaptivity curve to shift up and out, providing improved quality of solution for any particular point in the problem complexity space, as suggested by the third line in the figure.

3.6.1.8.2 Response Time

This refers to relative response time after some period of experience/learning. Response times should generally improve with learning (trending towards some absolute baseline).

3.6.1.8.3 Cognitive Cycles/Cognitive Operations

This is a measure of the cycles (or percentage of total CPU time) devoted to cognitive operations after some period of learning. COPS should not degrade significantly during learning or following it (e.g., due to significantly increased knowledge bases).

4. Experiments, Results and Discussions

To investigate the characteristics of the designed cognitive architecture, we applied PCAA Cognitive Layer to two problems.

4.1 Experimental PCAA Cognitive Layer Infrastructure

The Cognitive Layer team developed infrastructure needed to support experimental implementation of the demonstration problems (Figure 34). This infrastructure consisted of the following main components:

- **Proto Cognition**: Swarming algorithms were used to implement Proto Cognition.
- **Micro Cognition**: The ACT-R software was used as the implementation of Micro Cognition.
- **Macro Cognition**: The Soar software was used as the implementation of Macro Cognition.
- **mCML**: The interface API between components was the mCML XML format. Operations between components were encoded as mCML message types, and all data used in the demo problems was encoded as mCML chunks.
- **Messaging layer**: The XmlBlaster software served as both the messaging subsystem and the central repository for persistent data.
  - **Interface**: Custom software was written to make it easier for the cognitive component implementations to use the messaging layer. This software, called "CmlClient," mediated between the XmlBlaster messaging Protocol and higher level functions including principally the mCML operations.

Error!

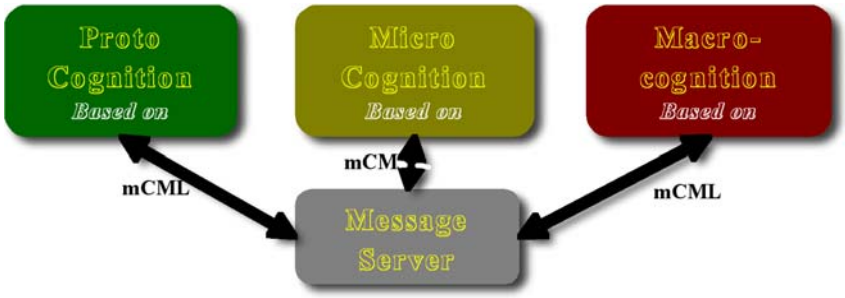


Figure 34. The Cognitive Layer infrastructure needed to support experimental implementation of the demonstration problems

4.2 Experimental Target Problems

4.2.1 Sign of Crescent (SoC)

The Sign of the Crescent (SoC) was the first of the two demo applications developed during the project. SoC was an artificial problem created to help train human analysts at the US Joint Military Intelligence College. The SoC problem has the following features of interest:

- Involves learning to marshal dynamic, incremental evidences ("trifles") to recognize a coordinated terrorist plot.
- Evidences are scattered spatially and temporally.
- Problem includes distractors – intelligence reports that have nothing to do with the plot

4.2.1.1 The SoC Problem

We applied the Cognitive Layer to help solve a portion of SoC. Specifically, in the SoC scenario, three near-simultaneous terrorist acts are being planned:

- A dirty-bomb explosion aboard a ship in Boston harbor
- A large bomb explosion aboard an Amtrak train (named "Crescent") in Atlanta
- A large bomb explosion inside NYSE

The SoC problem is an instance of plan recognition: Infer planned intention(s) from observed relevant and irrelevant actions. In particular, it is an instance of Keyhole plan recognition – actors are unaware of (and neutral about) the plan recognition process. It is also a hard problem, with algorithmic complexity ranging from polynomial (top down, cfg) to exponential (bottom up).

In intelligence jargon, the evidence marshalling process is called *Wigmorean Analysis* [Anderson05]. A sample Wigmorean analysis for a portion of the SoC problem is shown in Figure 35.

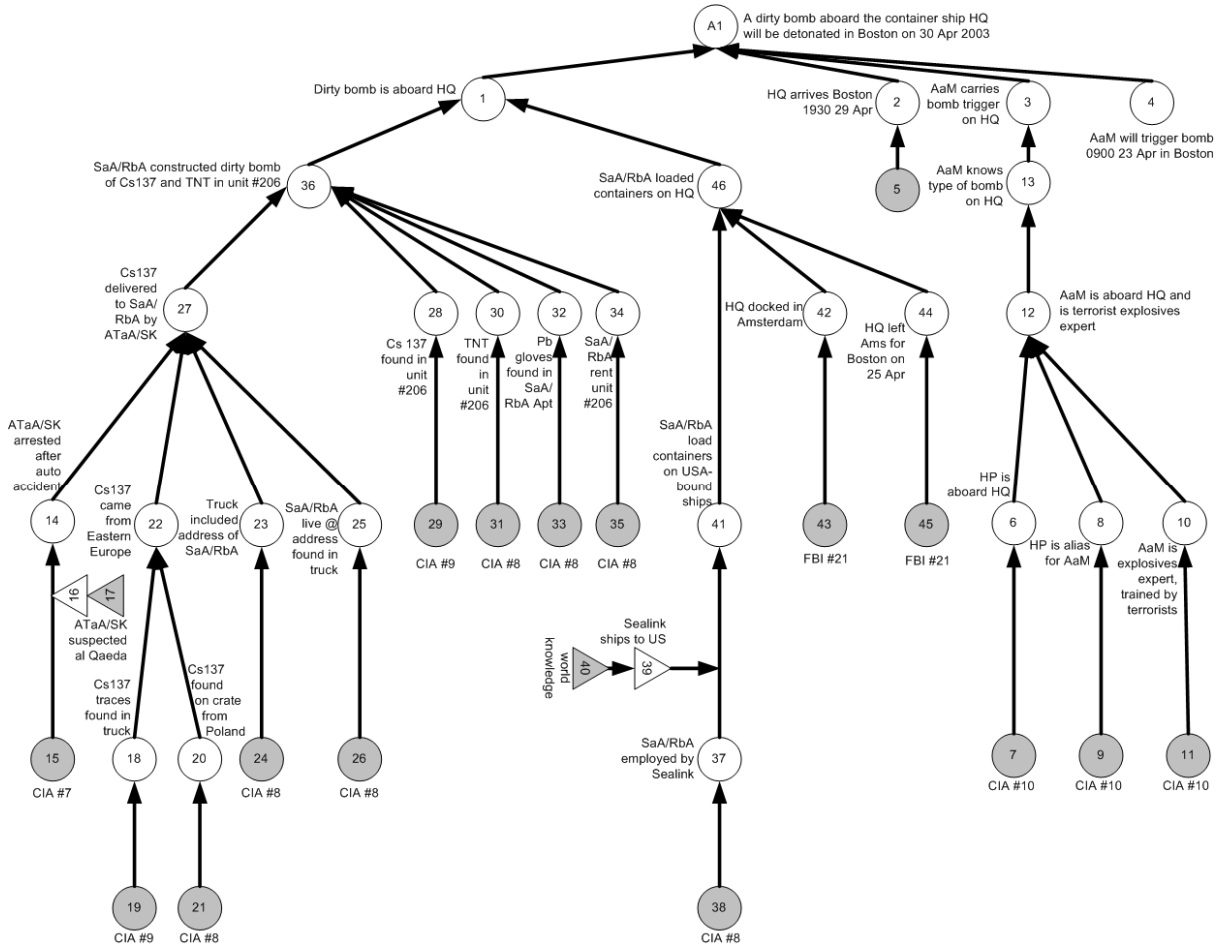


Figure 35. Wigmorean analysis of Sign of the Crescent

We applied our C3I1 cognitive architecture to solve a portion of the SoC problem by encoding the problem in mCML and extending the cognitive components (Macro/Micro/Proto), as necessary, for the problem. The result was rendered in a GUI developed for this purpose (Figure 36).

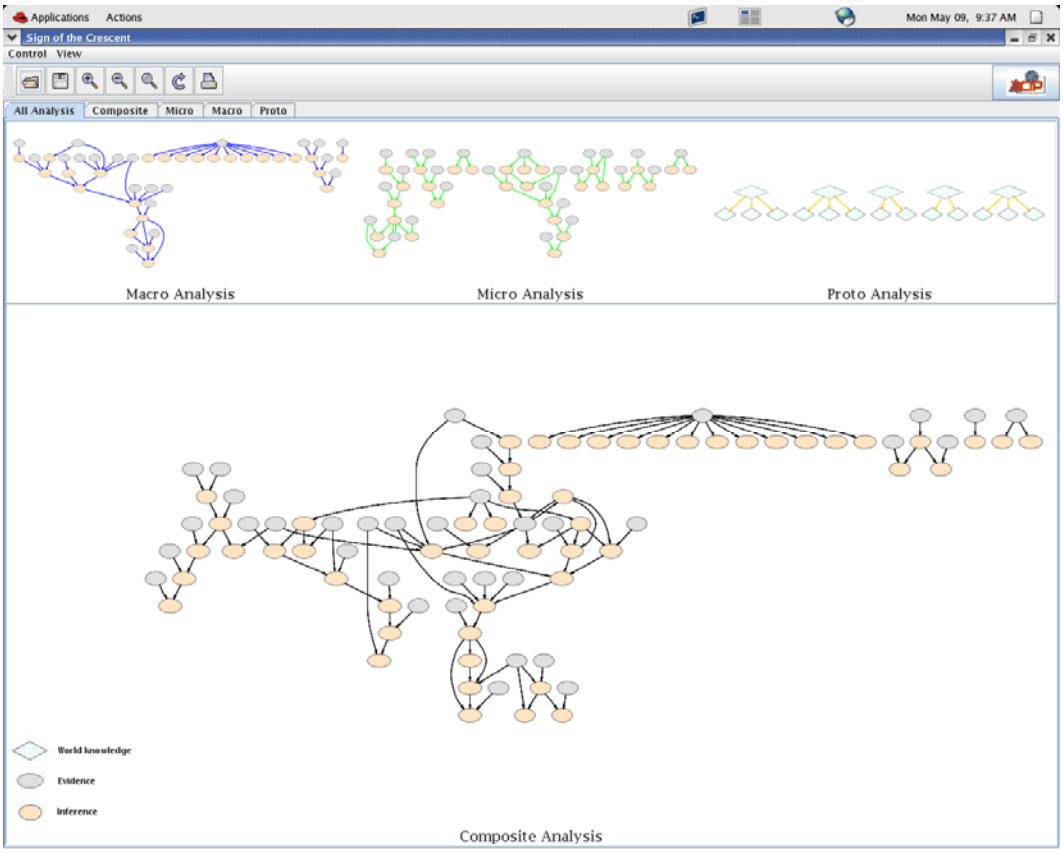


Figure 36. Sign of the Crescent demo problem GUI

4.2.1.2 Cognitive Solution Architecture

The three components of the C3I1 Cog structure adopted specific roles in the Sign of the Crescent problem.

4.2.1.2.1 Proto Processing (Figure 37)

- Similarity clustering to organize the data space
  - Hierarchical Ant Clustering (HAC)
    - Every node is an ant – *local* operations: promote and merge
  - Lexical association by Fair Isaac
- Compositional linking to generate solutions
  - Leverages similarity clustering and relational structure of data "events" (agent-action-object tuples)
  - Solutions are rough and partial

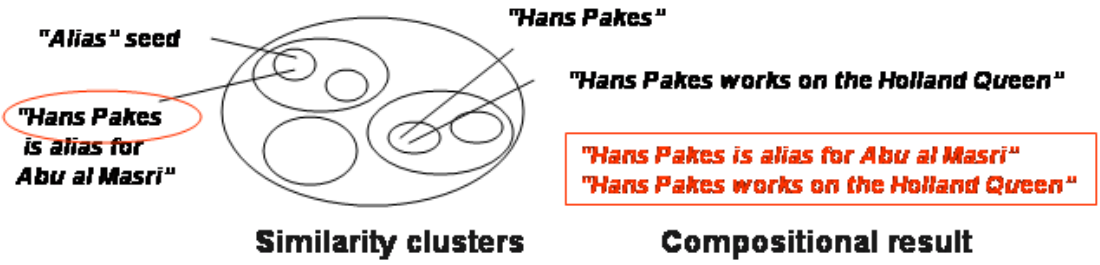


Figure 37. Proto Processing provides similarity clustering and compositional linking to generate solutions

4.2.1.2.2 Micro (Figure 38)

- Evidence marshaling as generalization of expert inferences
  - — Represent expert inferences in declarative memory
  - — Use associative priming to determine most useful inference(s)
  - — Similarity-based generalization of inference to current situation
- Interaction with other cognitive levels
  - — Clustering from Proto defines the context driving associative priming to select candidate inference
  - — Deliberative reasoning from Macro provides new inferences when existing knowledge is not relevant

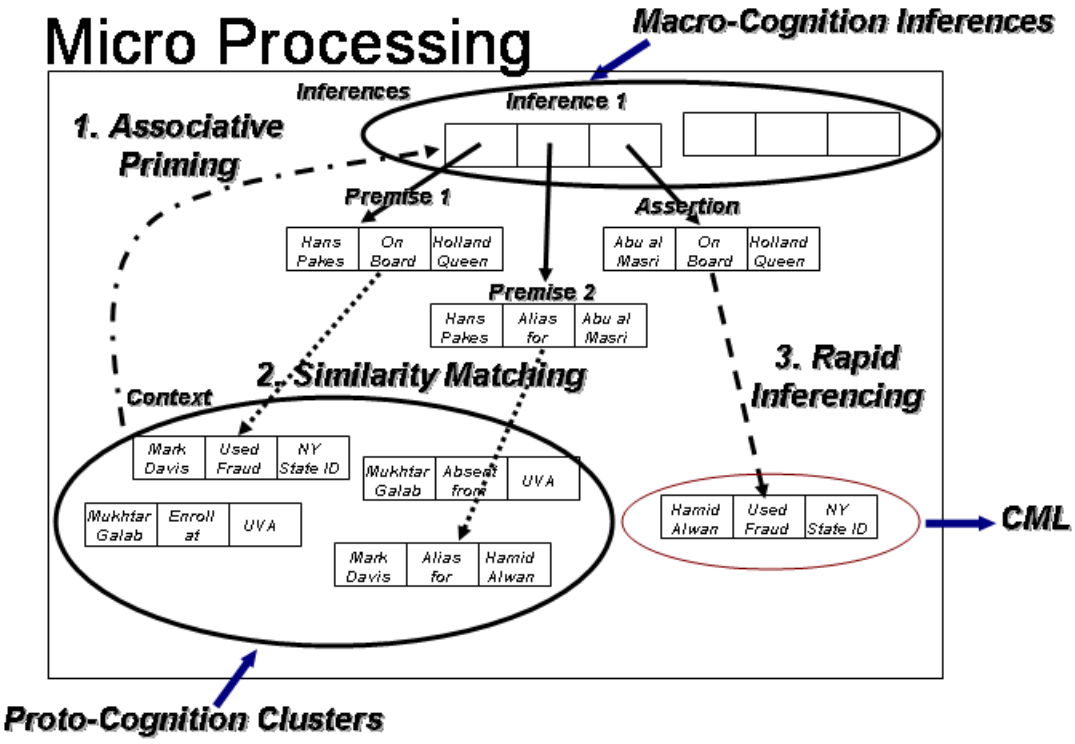


Figure 38. Micro Processing provides evidence marshaling and interaction with other cognitive levels

4.2.1.2.3 Macro (Figure 39)

- Establish premises (micro invocation)
- Retrieve directly related facts from C3I1 memory
- Generate new inferences (apply knowledge sources)
- Assess relevance of new assertions
  - Evaluation approaches
    - Analogical mapping
    - Social network analysis
- Augment C3I1 memory with relevant inferences

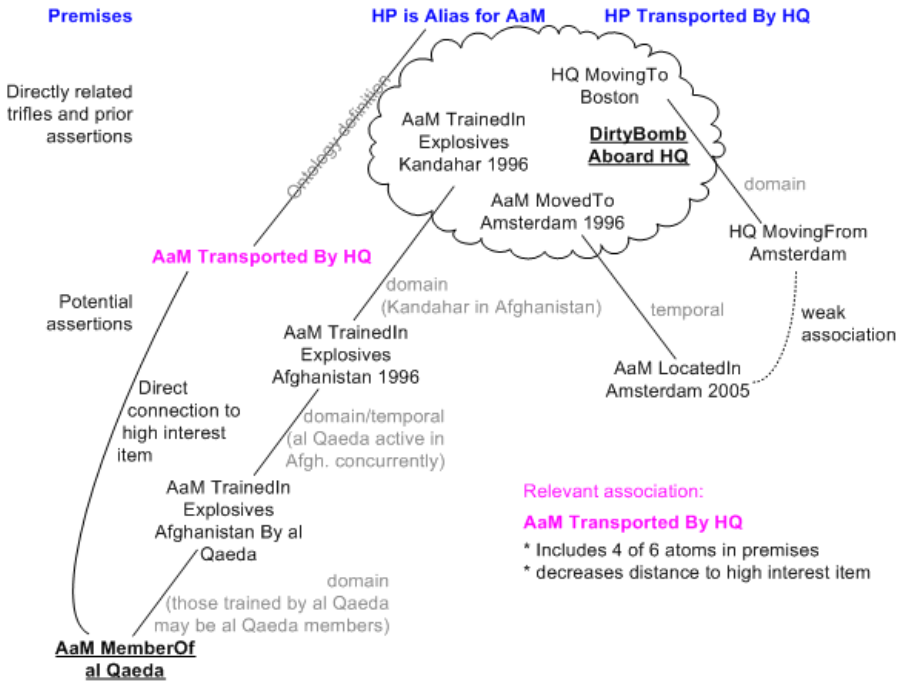


Figure 39. Macro Processing establishes premises and retrieves directly related facts from the C3I1 memory

4.2.1.3 SoC mCML Specification

The mCML types (XML Schema) for the Sign of the Crescent problem domain is given in the Section 7.4.1, due to length. Note that this schema is not the actual problem itself; that would be mCML (XML data) giving the particular data for a problem to be solved.

4.2.2 UAV Mission Planning (UMP)

UAV Mission Planning (UMP), in general, is a large problem that potentially includes many tasks such as route planning, task allocation, scheduling, and collaboration. For example, an overview of potential UAV mission planning tasks (as developed by the PCAA App layer team) is shown in Figure 40.

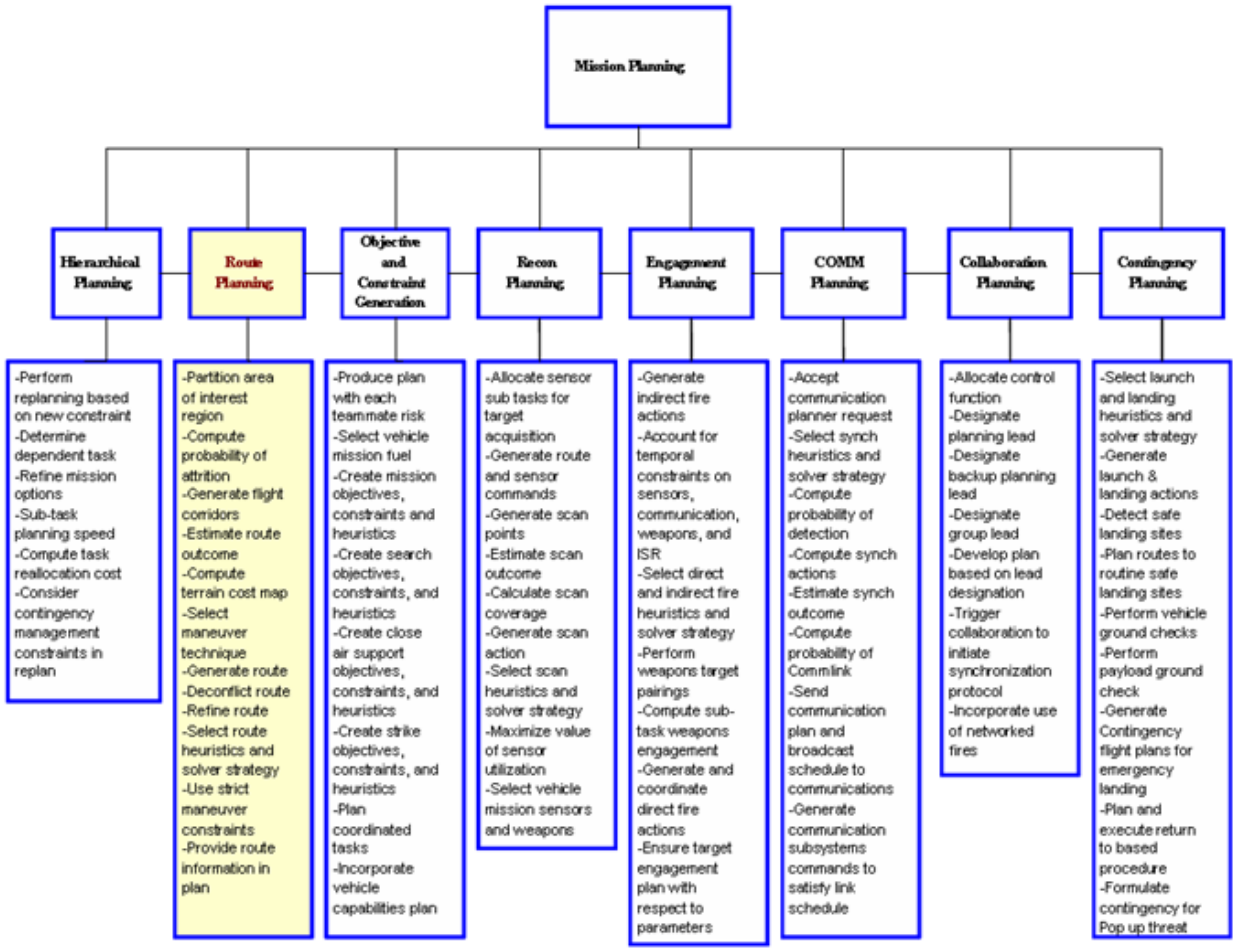


Figure 40. An overview of potential UAV mission planning tasks

For our PCAA demo, we focused on a single-UAV route planning task. We developed a specific scenario for this task, called Lightning Bolt. Scenario Lightning Bolt is given in its entirety in the next section.

4.2.2.1 Scenario Lightening Bolt

The U.S. has recently discovered a terrorist cell in a Middle Eastern country known to be hostile to the U.S. Because an intelligence agency has confirmation that a high valued target is currently located within the terrorist cell, the President has authorized the neutralization of this threat. Due to the sensitivity of the terrorist cell location in the hostile country, the President has asked that knowledge of this neutralization be minimized. As such, a highly survivable Unmanned Air Vehicles (UAV) will be used by the USAF to engage this target. Because of the high valued target is highly mobile, there is urgency for the mission and consequently the USAF signifies the mission as a Time-Critical-Target. Time is of the essence. In addition, the cell operates out of several locations. Although the high valued target can be expected to be in one of these locations, a coordinated assault on all of them is necessary to maximize the chances of neutralizing the target.

U.S. Joint Forces Command (JFCOM) assigns Lightening Bolt to the U.S. Combined Air Operations Center (CAOC) at Prince Sultan Air Base in Saudi Arabia for execution. After a quick evaluation of aircraft in the theater, their capabilities, and their availability, using a collaborative planning capability the CAOC has selected three classified USAF extremely autonomous Unmanned Combat Air Vehicle (UCAV) and assigned the assets with the code name Thumper. Thumper has a highly advanced Synthetic Aperture Radar (SAR) as its sensor, has Electronic Attack (EA) capability, and carries two 1,000-pound Joint Direct Attack Munitions (JDAM). JDAMs have high accuracy, all-weather, autonomous, conventional bombing capability. In this case the weapon is the GBU-32. The collaborative planning system has evaluated the alternatives and arrived at the solution that Thumper's capabilities are a perfect match for the mission Lightening Bolt mission (lethality evaluation). At the CAOC, the team generates an Air Tasking Orders (ATO) for the mission and the ATO is routed to Thumper's ground station controllers.

Only a few minutes have passed since the President gave his order, but it is early evening at Thumper's classified deployed location. Thumper's Electronic War Officer (EWO) immediately starts the mission preparation while ground maintenance personnel finalize the three aircraft for flight. Because of the number of targets within the cell, three Thumpers will be launched, Thumper-One, Thumper-Two, and Thumper-Three. The mission will use the cover of night and exploit an early morning engagement, when the defenders are the sleepiest. The Commanding Officer orders that the weapons be placed on the various targets at 0300 the next morning (thus assigning a time-over-target requirement), thereby minimizing the opportunity for various elements of the terrorist cell to warn each other of an attack. There are no restrictions on fly zones or restricted zones placed upon the mission. However, in attempt to limit the political fallout of the mission, the President has asked that no other country's political boundaries be compromised.

Although the terrorist cell is lightly defended, the hostile country is highly defended and is predicted to make every attempt to shoot down the UCAV and then parade the wreckage within the international press. The dynamically generated mission plan results in the UCAVs entering the hostile country from the southern coastal border, flying north several hundred miles to the target, maneuvering around the threats, and exiting the hostile country to the southeast. Four hundred miles of the mission will be over mountainous terrain. A critical component to the mission will be the use of terrain masking and the minimization of threat exposure, balanced with the UCAVs range (a factor of the amount of fuel that is available). The planning capability used as much of the mountainous terrain as possible to minimize threat radar acquisition exploiting the hostile sensor capabilities and line-of-site characteristics.

The regional weather is analyzed and predicted favorable over the target, but a storm is brewing in the Gulf of Oman heading northeast. The Thumpers will fight a head wind on their way back to the base, a concern to the EWOs as fuel is predicted to be critical at that point in the mission.

Because of the time-critical-target nature of the mission, the UCAVs are immediately fueled, mission data loaded, crypto keys loaded and the UCAVs are launched. (The reader should note that other scenarios may include redirection or tasking of aircraft that are already airborne. In this

case the planning is the same, but accomplished while the vehicle is airborne.) The Thumper controllers will use the on-board PCAA hardware to determine the mission, real-time, while they are in flight. The controllers provide an initial waypoint over the Persian Gulf and the UCAVs are on their way. During the first few moments of their flight, the Low-Probability of Intercept (LPI) Low Observable Spread Spectrum Frequency Hopping Satellite Communication (SATCOM) is used to load the known hostile country's Surface-to-Air Missile (SAM) batteries (threats to Thumper) to the PCAA memory. At the same time, the EWOs have analyzed the terrorist cell intelligence data from information provided by the intelligence agency and have determined five target-coordinates, Designated Mean Point of Impact (DMPI).

4.2.2.1.1 Initial Planning

At this point, while en route to the initial waypoint over the Persian Gulf, the PCAA system analyzes the mission and determines an initial assignment of DMPIs to the individual UCAVs. This assignment takes into account the value of the various targets, the capability of the assets to fulfill the mission (range, munitions), known threats in the target area, and the potential impact of losing one or more of the available UCAV assets on the overall plan. After determining a plan to accomplish these goals, the information is transmitted to the UCAVs, with each Thumper getting at most two DMPIs, while one of them will get the fifth DMPI (all five DMPIs will be assigned).

Each Thumper UCAV then proceeds to calculate their specific portion of the mission in greater detail independently. The area where the Thumpers will be flying has a large threat density with many of the SAMs containing overlapping coverage. Worse, several identified SAM sites are thought to be housing a new type of device not well known to the U.S., thus the threat's coverage area capability provided to Thumper's PCAA was biased on the worst case, adding complexity in determining the optimized route. The UCAVs also calculate the Launch Acceptable Region (LAR) for each DMPI and integrate those constraints into the solution space. The PCAA system will also calculate the optimum location where expendable jammers are to be dropped. These locations are provided to the mission router as route objectives, a location for the aircraft to perform the EA function. Additionally, each aircraft already has its Radar Cross Section (RCS) templates loaded for each threat in the hostile country's inventory (and a few more just in case). The RCS templates will be used in the route determination process in an attempt, by using Thumper's low observable properties, to avoid a threats ability to detect the aircraft, at all look angles (the minimization of threat exposure). The use of these templates will dictate flight qualities such as latitude, longitude, elevation, roll, pitch, and yaw.

4.2.2.1.2 Planning Deconfliction

After just a few elapsed seconds, the UCAVs have calculated a route to the target and transmit their routes to each other. This starts the route deconfliction process, during which the Thumper assets negotiate their plans collaboratively. From the weapon drop location within the LARs, the weapon fly-outs are calculated. Each Thumper asset will attempt to avoid weapon fly-out zones for the other UCAVs. For example, Thumper-One may discover that the second weapon launched from Thumper-Two is dangerously close to Thumper-One's projected egress route and

accordingly determine an alternate weapon launch point within the LAR that eliminates the danger. Additionally, the deconfliction process will adjust the routes used within the target area so that each aircraft is attacking the target from different directions. And likewise, the flight over the valley is optimized so that each aircraft is deploying jammers in locations that attempt to optimize EA for both aircraft. Finally, the UCAVs will ensure that their egress and ingress routes do not overlap. For example, if Thumper One uses an egress route that is identical to the ingress route used by Thumper Two, this will result in the same territory being overflown twice, increasing the risk of hostile action against Thumper One. The deconfliction process will result in recalculation and adjustment of routes based on these additional constraints. Each UCAV transmits the finalized mission to the ground controllers and EWOs.

4.2.2.1.3 Pop-Up Threat Example

The Thumper UCAVs proceeded with their mission, flying into the hostile country undetected, traversing through the large mountain range. At the same time, a high altitude Unmanned Reconnaissance Air Vehicle (URAV) flying over the Persian Gulf, monitoring the mission detected a previously unidentified SAM site with very lethal consequences. Because the URAV has Network Centric Communication (NCC) abilities, the URAV immediately transmits the new finding out on the network. Our Thumper aircraft monitoring the network automatically add the newly discovered threat into their database and begin recalculating the route. However, unlike the other additions discovered this night, this threat requires avoidance. The UCAVs recalculate their new routes to avoid the threat. In doing so, Thumper One identifies an inability to complete its mission objectives and successfully return to base. As a result, it offloads its most distant target to Thumper Three, while Thumper Two replans its route to avoid conflicts with the new routes chosen by Thumper One and Thumper Three.

4.2.2.1.4 Example of Pop-Up Threat using a Cognitive Solution

The threat intelligence gathered for the area where the UCAVs are flying is limited. As the mission Lightening Bolt continues, the URAV detects yet another previously unidentified SAM site with very lethal consequences. Once again the Thumper aircraft are notified of the threat and in the process of regenerating a route that maximized his stealthy characteristics to best avoid the threat. However, the new route due to this threat will cause the mission constraint of the 0300 Time-Over-Target (TOT) to be missed. Because mission Lightening Bolt also has special forces ground troops attacking the target at 0305, the TOT was predetermined to be the highest priority constraint, because this constraint can not be violated due to fear of fratricide. The UCAVs now must replan using the mission constraint causing the generated mission plan to incur detection time, yellow time, and small amounts of lethal, red time (time when a threat can shoot down the vehicle). Due to the sensitivity of the mission, the planning system must minimize the red time and the corresponding yellow time to ensure minimal opportunity for the enemy to detect Thumper and become alerted to the mission. The collaborative planning system is alerted to the incurred red time and yellow time and a collaborative support action of electronic attack is generated for the Thumper team thus mitigating the potential red time in the mission. A collaborative electronic attack method is used to further minimize to probability of detection.

Thus, the planning suite has found a plan that eliminates red time, yellow time and makes the TOT. The two UCAVs make the necessary course changes and continue the mission.

4.2.2.1.5 Example of Target-of-Opportunity Exploitation

While proceeding to target, one of the Thumper assets passes near a previously unreported camp site in a vicinity known to be a hotbed of terrorist activity. The team at the CAOC determines that this site is the likely location of another high priority target and retasks the PCAA system with integrating a strike of this target with the original mission plan. Accordingly, the Thumper asset that is best able to attack this target is assigned to the new high priority target, and the other Thumper assets are reassigned as necessary to achieve as many of the mission goals as possible.

4.2.2.1.6 Example of Weather-Based In-Flight Replanning

An unexpectedly strong head wind is produced from the storm in Gulf of Oman. As a result, there is not enough fuel to complete all mission objectives and return to base. The PCAA system must recognize the progress being made against the mission plan and identify the likely mission failure. Taking this into account, the replanning process is engaged, once again evaluating the known targets, threats, and available assets. For example, Thumper Two may encounter severe weather along its route, while Thumper One and Three avoided the worst of the weather. At this point, given Thumper Two's now reduced endurance, Thumper One and Three must redirect their resources to help accomplish the overall mission objective. The PCAA system recognizes this, assigning one of Thumper Two's previous DMPIs to Thumper Three.

4.2.2.1.7 Example of Equipment Failure-Based Replanning

During flight, Thumper Three experiences a failure of its onboard GPS system, and as a result, Thumper Three will be unlikely to reach its DMPI. The PCAA system reacts to the failure by attempting to return Thumper Three to the base area (through compass navigation), or at least to friendly territory. Simultaneously, the PCAA system replans the mission given the original mission targets and the current (reduced) assets. At this point, only four of the DMPIs can be targeted, so the PCAA system engages in replanning based on balancing the value of the individual targets with the likelihood of successfully striking those targets.

4.2.2.1.8 Example of Intelligence-Based In-Flight Replanning

When Thumper-One and Thumper-Two are only about 100NM from the target, a space based national asset detects that the terrorist have suspected the attack and are now leaving in a vehicle, at high speed, to the northeast following a dirt road. JFCOM immediately designates the vehicle as a target, or DMPI. The COAC assigns Thumper-One to attack the moving vehicle with one of its weapons and then with the remaining weapon attack the terrorist cell. In doing so, it is desired to have Thumper-Two wait for Thumper-One to attack the terrorist cell. The terrorist cell DMPI priorities are recalculated and adjustments are made so that the top three DMPIs are hit. (It is possible that Thumper-One had DMPI priority one and two. When Thumper-One is reassigned to drop one of its two weapons on the terrorist fleeing in the vehicle, the second highest priority

DMPI is now not covered. Because Thumper-Two has the third and forth highest DMPI priorities, tactics dictates that Thumper-Two delete the fourth highest target and accepts the second highest priority resulting in attacking the second and third highest priority DMPIs with the fourth highest target left unengaged.) Both aircraft repeat some of the same mission planning process that was performed initially, to target both the vehicle and the terrorist cell. During the attack, SAR imagery is captured and sent to the EWOs for battle damage assessment.

The ideal result from this scenario is to have all targets neutralized with all aircraft returning safely to base. Additionally, it is desirable to have had the mission executed clandestinely with no traceability to the U.S. and no information arriving in the press.

4.2.2.1.9 Issues

Replanning is a central challenge of the domain. How centralized should this be? The original scenario placed the locus of planning control with the PCAA system during the early mission, and with Thumper One during the mission execution. This seems like a design decision rather than an aspect of the scenario.

Communication during replanning is essential, but can we depend on it? Communication between UAVs can alert the enemy to their presence, and in addition is unreliable. What if a UAV loses its ability to communicate? Should it continue its mission?

4.2.2.2 UMP Scenario Implementation

The Lightning Bolt scenario was refined down to a few simple, concrete components for use as a demo exercise by the Cog layer. These components were:

- **Map**: The scenario was defined to take place on a 2D map overlaid with a 100 by 100 coordinate grid. The positions of other components were specified in terms of this grid.
- **UAV**: The UAV itself was an implicit component of the scenario.
- **Targets**: A set of target locations was defined on the map. The UAV had to visit all targets, in an order of its choosing.
- **Threats**: A set of threat locations was defined on the map, each with a threat radius. The UAV had to complete its tour without entering any threat radius.

As defined, this scenario is very similar to the classic Traveling Salesman Problem (TSP). However, it was noted that this problem has a strong cognitive aspect, and the goal was to focus on this aspect rather than to find a possibly quicker non-cognitive solution.

Additionally, the scenario was designed to support pop-up threats, which are threats that appear on the map after the UAV is in flight executing its original plan.

4.2.2.3 Cog Solution Architecture

The three components of the C3I1 Cog structure adopted specific roles in the UMP problem:

- Proto cognition provides clustering and grouping services.

- Microcognition provides focused local planning and execution
- Macro Cognition provides planning services

The development of the route plan proceeds along the following lines (Figure 41).

1. **Proto performs hierarchical clustering (Figure 41a).**

Blue = target
Red = threat
Yellow = Other

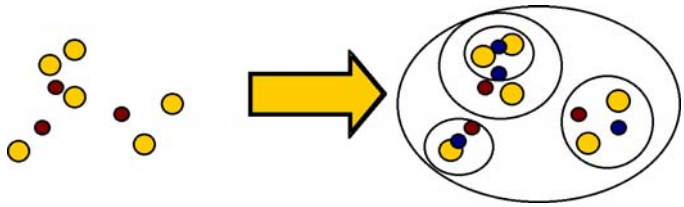


Figure 41a. Hierarchical clustering

2. **Proto finds best paths between targets in cluster (Figure 41b).**

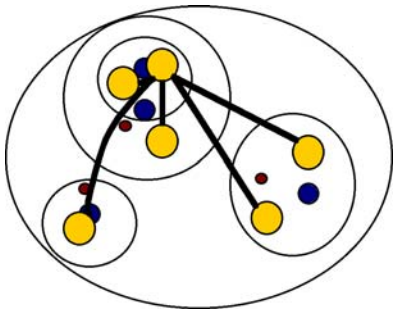


Figure 41b. Path identification

3. **Micro finds best tours between clusters and within-cluster targets (Figure 41c).**

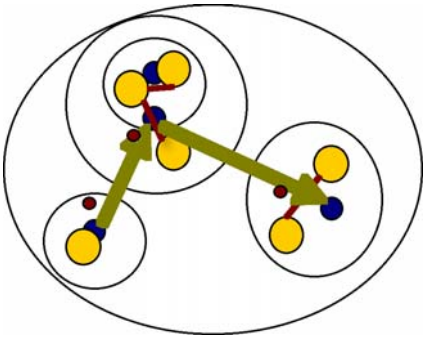


Figure 41c. Micro finds best tours

4. **Macro finds best tours for any remaining more complex cluster sequences (Figure 41d).**

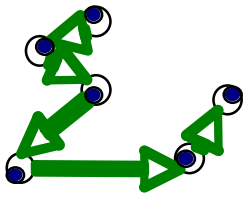


Figure 41d. Macro finds best tours for any remaining more complex cluster sequences

4.2.2.4 UMP Demo Example

In this section we go over an execution of the demo architecture on a very simple form of the sample problem, having seven targets and only one threat (Figure 42 - 50).

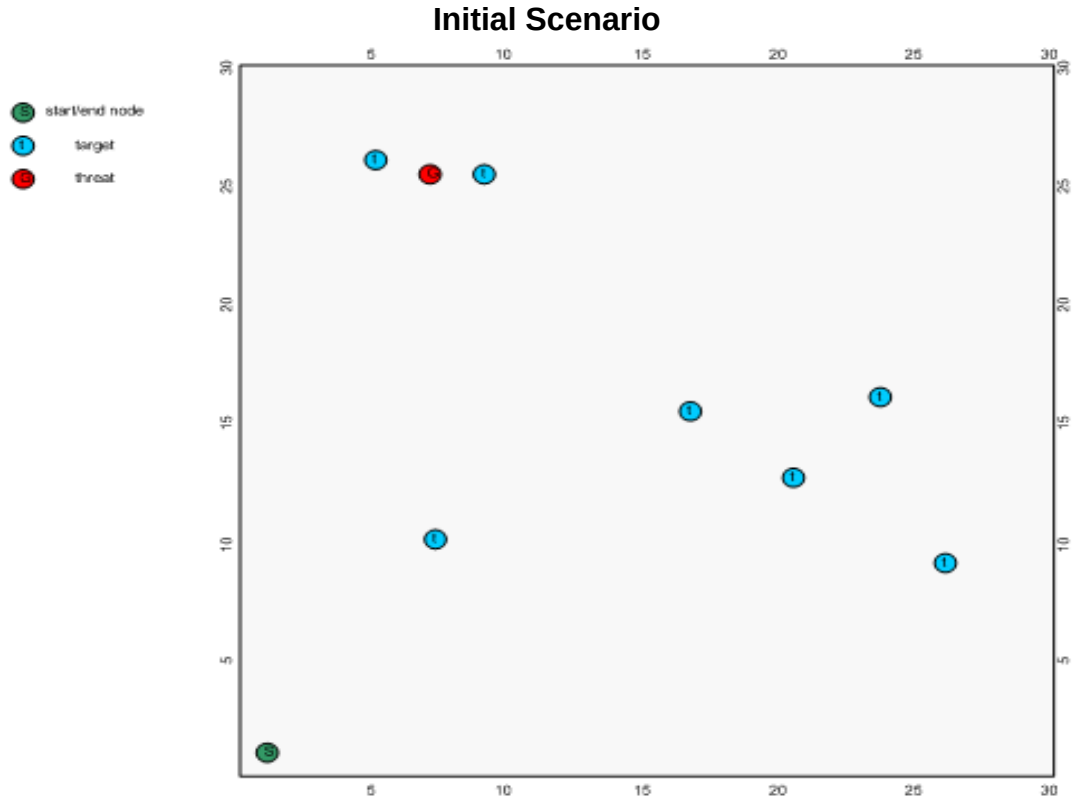


Figure 42. The initial map, showing the UAV's starting location in the lower left, the seven target locations, and the threat location.

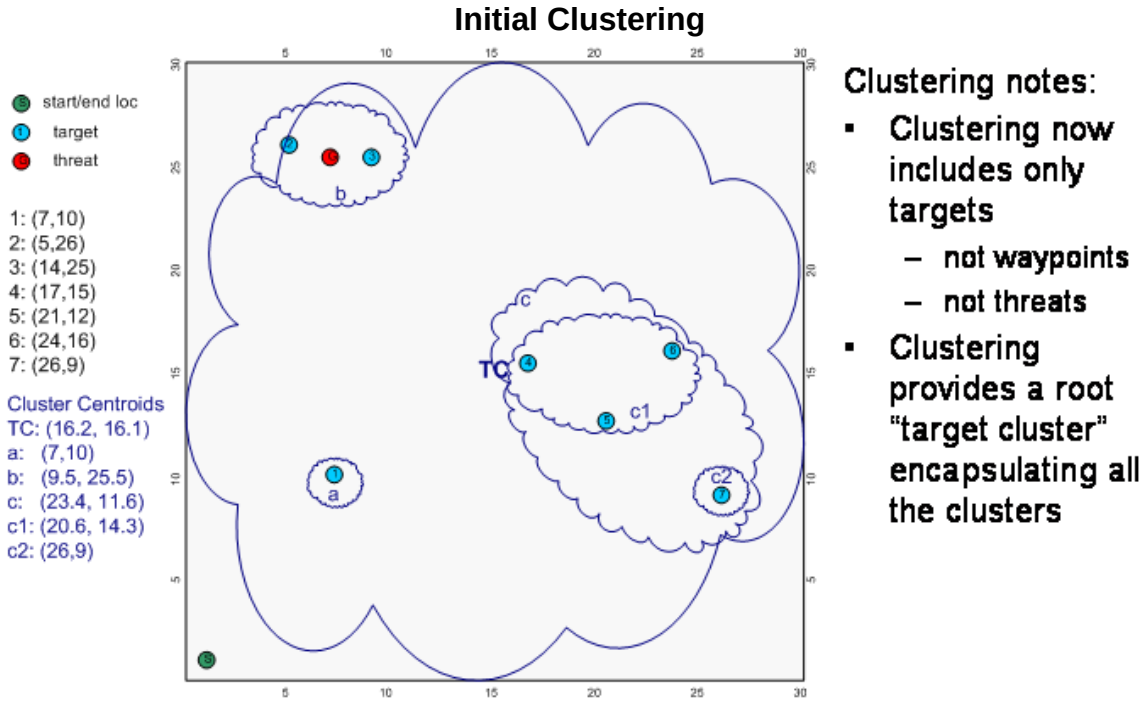


Figure 43. Initially, Proto cognition clusters the targets.

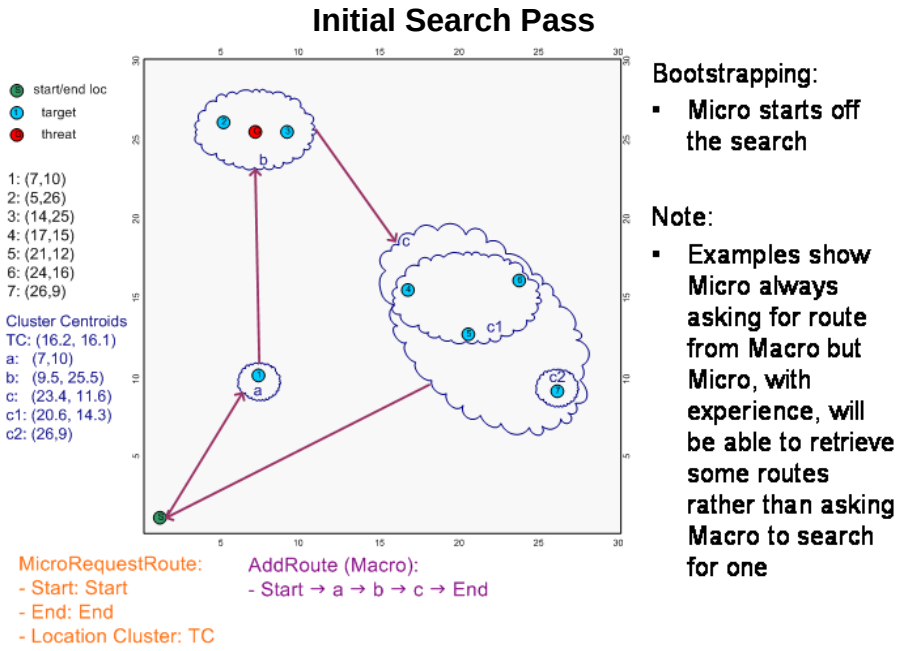


Figure 44. Micro attempts to construct a coarse route sequencing the clusters themselves. For expository purposes, in this example Micro is unable to find a route in it store of experience, so it asks Macro to construct a fresh route. Micro learns the route and will be able to reuse it in future cases.

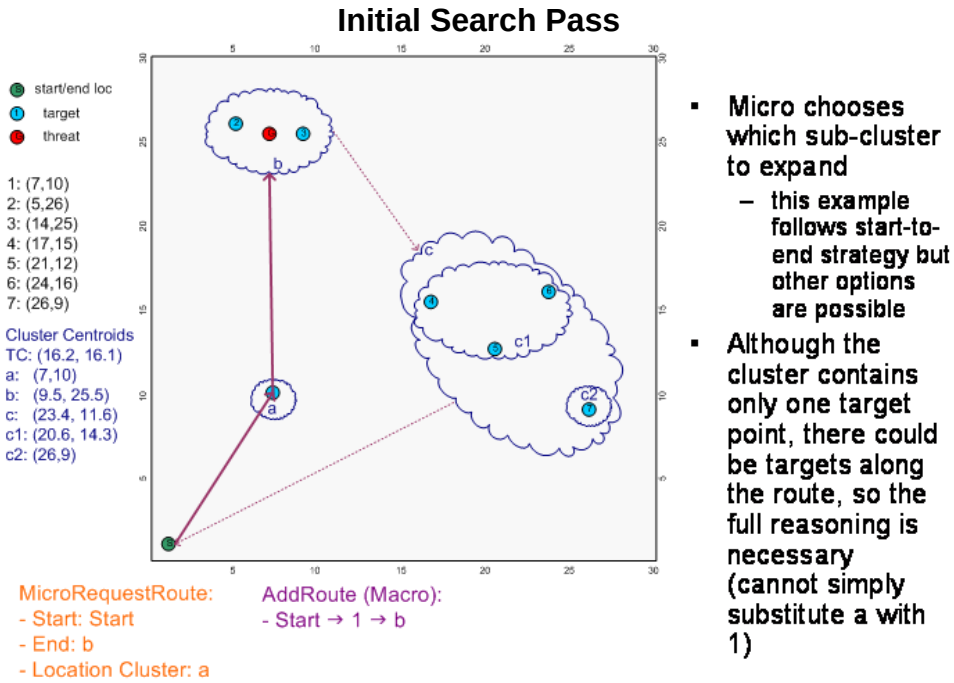


Figure 45. Having a top-level route, Micro must now delve into the clusters, forming intra-cluster routes. The first cluster is trivial, containing only one target.

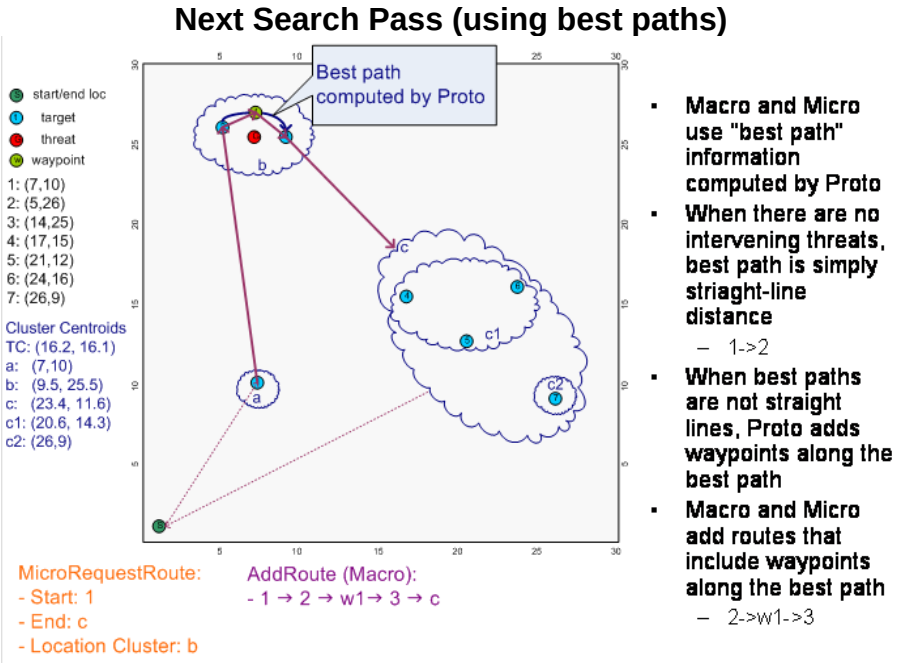


Figure 46. Micro continues finding paths within clusters. The second cluster is more complex. However, Micro is able to use a simple path called the "best path" that is constructed by Proto.

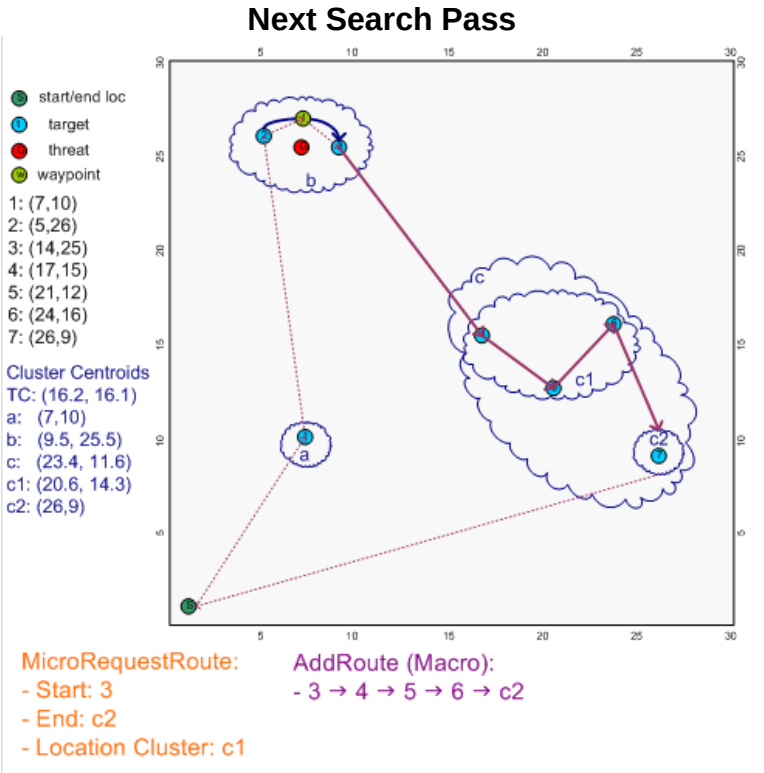


Figure 47. Going to the next cluster, Macro is invoked again to find a route.

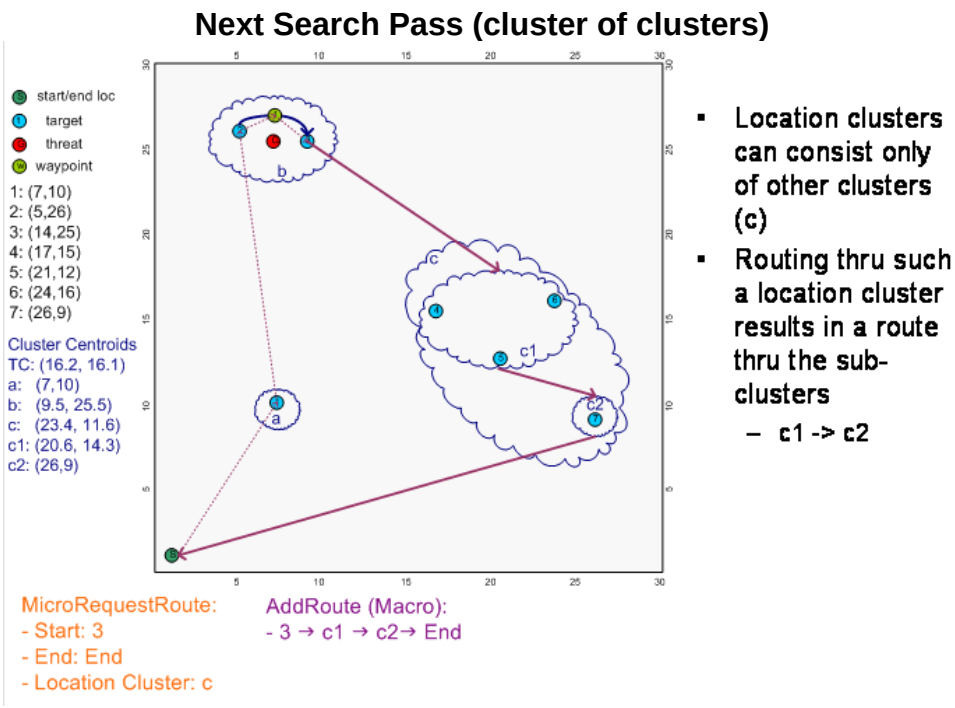


Figure 48. This cluster offers an additional complexity in that it contains child clusters. In such cases, the route is specified as leading in and out of the cluster, but the specific endpoints within the inner clusters are left unspecified.

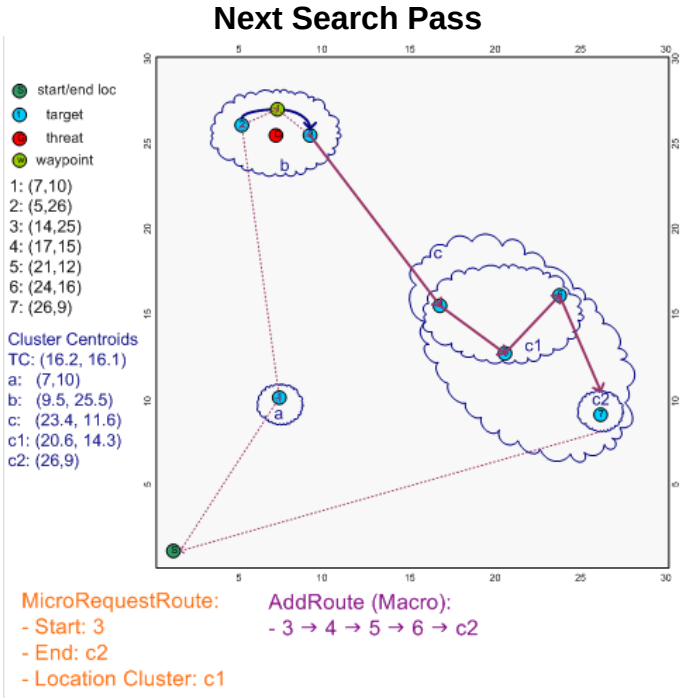


Figure 49. The routes through the inner clusters are filled out by appealing to Macro. This implicitly selects the endpoints for connection to the targets in other clusters.

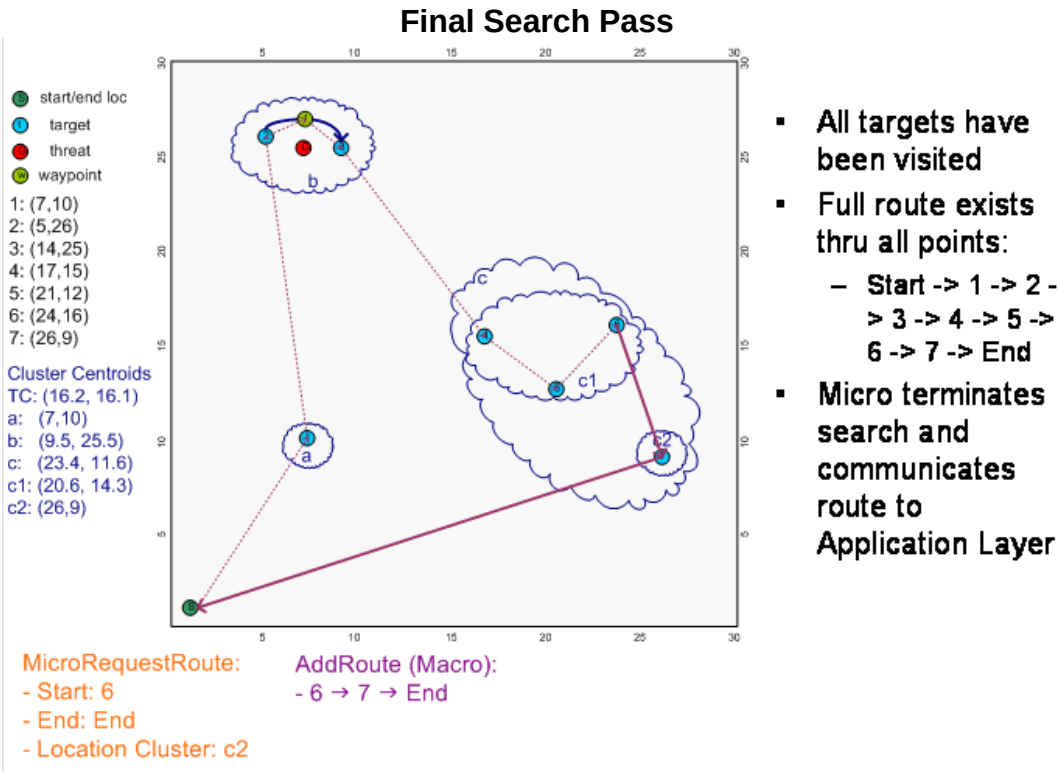


Figure 50. The final leg is added, and the path is complete. Micro recognizes that the path is complete and forwards the solution back to the Application layer.

4.2.2.5 UMP mCML Specification

The mCML types (XML Schema) for the UAV mission planning problem domain are given in the Section 7.4.2, due to length. As before, this is not the actual problem itself; that would be mCML (XML data) giving particular targets and threats for which a plan is needed.

5. Conclusion

Inspired by a model of human cognitive architecture, the PCAA Cognitive Layer (named C3I1) supports seamless processing of a greater range and scale of problems on multiple orthogonal dimensions: representational dimension (symbolic through subsymbolic), concurrency dimension (sequential through parallel), reasoning dimension (knowledge-based, expertise-based, and perception-inspired), and the dimension of problem-solving mechanisms (e.g., semantic data pyramid, iterative refinement). It provides an early prototype of such an architecture by seamlessly combining three heterogeneous components: Macro Cognition, Proto Cognition, and Micro Cognition. To estimate some characteristics of the architectures, we apply the prototypical cognitive architecture to two different, hard, experimental problems: an evidence marshaling (named "SoC") problem and a UAV mission planning (UMP) problem. Results from these experiments provide support for some of the design goals including:

a. **Generality**: C3I1 seeks to provide a very general computational cognitive substrate, enabling both high degrees of applicability and robustness. As a general architecture, C3I1 shows promise. The same core mechanisms and design demonstrated value in the highly symbolic, knowledge-intensive evidence marshalling domain, as well as in the more algorithmic, more dynamic routing problem. However, special-purpose mechanisms were developed within each layer for each application (e.g., in routing, threat-warped clustering, topologically-oriented memory retrieval, and the Traveling Salesman Problem (TSP) sorting heuristics). Our investigations, however, suggest a fixed, domain-independent set of mechanisms for Proto is feasible.
b. **Taskability**: C3I1 should not only apply to a broad range of problems, but it should do so without requiring extensive developer modification and have inherent ability to adapt to the requirements and constraints of particular problems. Taskability includes the ability to handle novel variations of problems and improve with experience. It also includes extensibility: the architecture should require small (and increasingly smaller) human effort to achieve good results in different applications. The taskability of C3I1 is both less clearly encouraging and more tentative. In the near-term, significant developer involvement will be needed in each layer; essentially, the approach requires a manual decomposition and mapping of the problem to the functionality of the three layers. Over time, we believe that tighter integration of the mechanisms will improve taskability, as a more constrained architecture will guide solutions more explicitly.
c. **Efficiency**: General, architectural solutions are always likely to be less efficient than special-purpose ones. To compensate, overall efficiency of C3I1, especially in the core loops of processing, is critical. Efficiency includes responsiveness to specific situations, overall resource management, and scalability across problems of increasing complexity. We have evaluated the computational bounds at each level – see Section 5.1.2. In terms of worst-case performance, the architecture should perform acceptably well, assuming specialized, parallel hardware for Proto and Micro. A key assumption is that Proto will reduce the overall scale of the problem by several orders of magnitude, in order to make Macro problem search generally tractable. Our initial explorations suggest Proto can accomplish a level filtering and aggregation that greatly reduces the raw scale of reasoning problems, but it is an open, empirical question how generally this power can be realized across domains and applications. Based on tentative results from the SoC and UMP experiments, we project (Figure 51, and

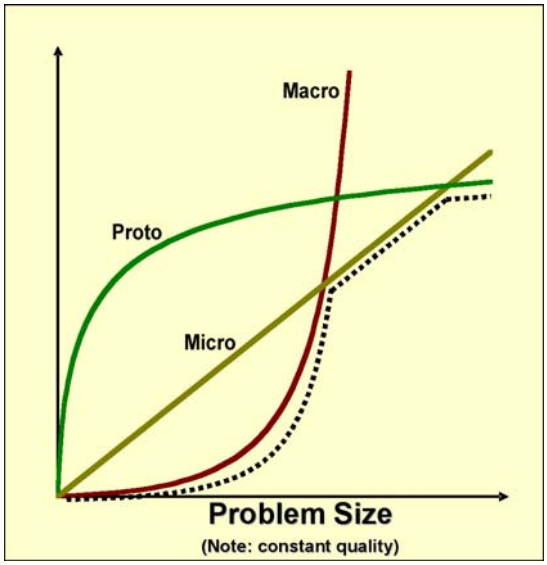


Figure 51. Problem-size vs cost for different cognitive components

the complexity analysis from Section 4.1) that the C3I1 architecture can solve hard problems in a scalable fashion.

While we think that the C3I1 architecture supports many other design goals too (e.g., adaptivity, robustness), we did not have the time or the resources to design and run appropriate experiments to show unambiguous support for those goals.

In terms of the methodological hypothesis, the explorations did identify potential opportunities for much tighter, more synergistic integrations of the components – more about it in Section 5.1 below.

Also, other alternatives are possible for the cognitions of C3I1. For example, neurologically-inspired clustering might be appropriate for perceptually-dominated domains. We discuss this issue in Section 5.3 below.

5.1 Complexity Analyses from the Cognitive Layer Components

5.1.1 Micro Cognition Complexity Analyses

The simulations results reported here focus on the impact of Proto focusing on micro efficiency for the model of evidence marshalling described elsewhere. There are other potential sources of additional efficiency, including hardware parallelization, but those will be examined separately.

5.1.1.1 Inferencing

The first step in Micro is selecting an inference to apply. Figure 52 presents results in terms of number of inferences that have to be attempted before the correct one is found.

The set of expert inferences is potentially quite large, e.g., in the thousands, so the X-axis plots the total number of inferences on a log (2) scale, basically from 1 to 1000. We will refer to that axis as the problem complexity, even though it has more to do with knowledge complexity than problem complexity. The number of inferences attempted will be referred to as the solution

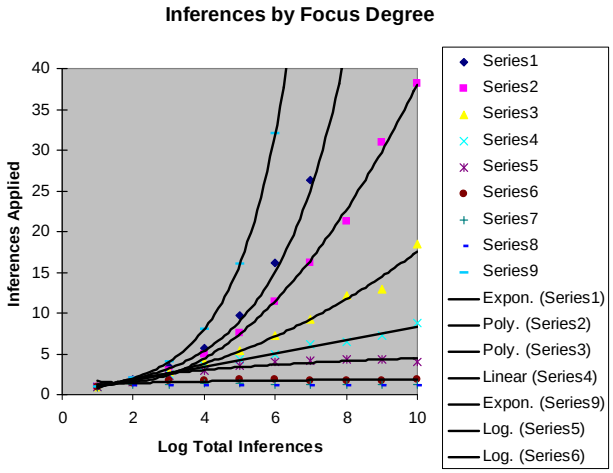


Figure 52. Inferences attempted as a function of Proto focus

complexity, even though there are additional factors to the solution complexity (e.g., the cost of applying it). Without the focusing provided by Proto, inferences will be selected at random, because no other basis in activation exists for selection than the activation's random component. This will lead to an exponential increase in number of attempts as a function of complexity. This is Series 9, and it can be seen as the baseline of applying Micro in isolation.

The primary function of Proto focusing on Micro is to provide a context made up of a cluster of trifle chunks to spread activation to related chunks, including other trifles and inferences. The result of that priming will be to raise the activation value of the most related chunks and depress the activation of the least related ones. The fundamental assumption of this simulation is that the distribution of activations follows a power law distribution, with a few very active chunks and the number of chunks with a given activation level increasing exponentially as the activation level decreases. There is very good support for that assumption, both specifically for the distribution of activations in ACT-R, and much more generally as the most common distribution in our cognitive environment (e.g., [Anderson91]). The key parameter is the slope of that distribution, i.e., how much more active are related chunks than unrelated ones. In other words, how precise is Proto's clustering in terms of pinpointing the correct inferences and trifles to focus on?

Figure 52 reports the results for a variety of slope values of the activation distribution. The proper way to interpret the slope is relative to the activation noise parameter, which can be interpreted as representing the factors beyond control. That parameter value is set to a consensus value of 0.25 used in many ACT-R models. One can see that for very low discrimination values, i.e., 0.1 (Series1), the solution complexity still grows exponentially with problem complexity, albeit more slowly than without Proto help. For slightly higher slope values, e.g., 0.15 (Series2) and 0.2 (Series3), the solution complexity improves to polynomial, with a good fit to quadratic growth. For a slope equal to the noise factor, i.e., 0.25 (Series4), the solution complexity is now linear in terms of problem complexity. For slightly higher discrimination slope values, e.g., 0.3 (Series5) and 0.5 (Series6), solution complexity grows even slower, i.e., logarithmically. For higher still slope values, e.g., 0.75 (Series7) and 1.0 (Series8), solution complexity grows very slowly and is almost independent of problem complexity. Of course, it might be that such discrimination might not be attainable in practice, because of limits on clustering quality, degree of predictiveness (i.e., Bayesian log likelihood) underlying the association strengths that carry spreading activation, or both. The key question is an empirical one: what degree of focusing in term of activation distribution toward the correct Micro inferences can Proto accomplish?

5.1.1.2 Generalization

The second step of the inference process after selecting an inference is to apply it. That means matching the inference premises to the contents of memory, and if successful making the assertion specified by the inference, else label it as failed and attempt another one (or give up and ask Macro, but this tradeoff will not be considered here). If this was an exact match, this would take place in a single step. However, the inference needs to be generalized from its original situation to the current one, and therefore the process will be an iterative one in which a match is attempted and may fail because some of the premises are not satisfied, then another one has to be

attempted until the right generalization is found (or abandoned and the process moves on to attempt another inference). This iteration is similar to the iterative process of finding the right inference, and the same analysis can be applied to it.

Two separate activation processes converge to narrow down the list of possible matching chunks to complete the generalization of the inference. The first one is the partial matching process that takes an inference premise and decrements the activation of the trifle chunks by their degree of (mis)match to the premise, based on the similarity between their constituent sloe values. We assume that process imposes on the activation of chunks a power law distribution similar to that for inferences, but this time controlled by a parameter reflecting the magnitude of the mismatch penalty, set at 0.15 to reflect common values. The top curve in Figure 53 (Series1) plots the average number of retrievals attempted per inference for this partial-matching generalization process, which grows as a polynomial (quadratic) function of the log number of chunks.

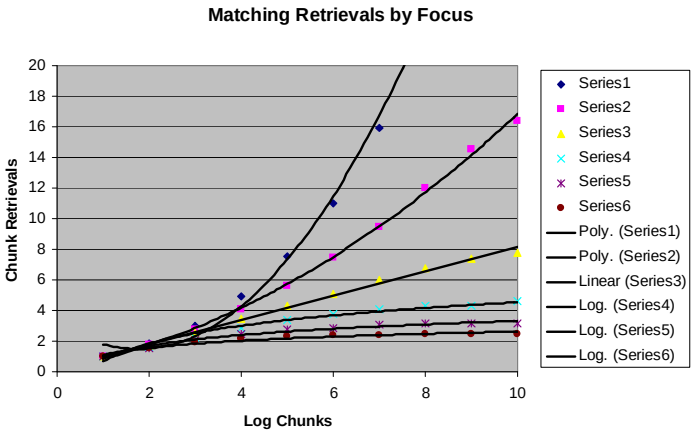


Figure 53. Trifle chunks retrieved per inference as a function of Proto focus

The second process controlling the activation of the trifle chunk is the same spreading activation process that governed the retrieval of inferences. As in the previous section, the result is a power law distribution favoring chunks related through strengths of associations to the cluster components. One would expect the two distributions, one generated by partial matching and the other by spreading activation, to be relatively orthogonal, with some chunks related to the cluster matching poorly the inference premises while some chunks that match the premises well are unrelated to the cluster chunks. However, for simplicity, we will assume here that the distributions are correlated, with the slopes of the distributions combining additively. Results for a spectrum of values of the focus slope similar to those in Figure 52 are presented in Figure 53 (the slope of the matching distribution is left constant at 0.15). For a value of 0.05 (Series2), the curve is still polynomial but already flattens considerably. For a relatively low value of 0.1 (Series3), the number of chunks retrieved per inference grows linearly with the log of the total number of chunks. As in Figure 52, this linearity results because the total distribution slope (0.1 for the focus activation and 0.15 for the matching process) is equal to the amplitude of the noise distribution (fixed at 0.25). For larger values of the focus slope, e.g., 0.15 (Series4), 0.2 (Series5) and 0.25 (Series6), the average number of chunks retrieved per inference grows logarithmically with the problem complexity, defined as the log of the total number of chunks.

5.1.1.3 Total Solution Complexity

Since memory chunk retrieval and production matching are the basic Micro cognitive operations, the total solution complexity should be expressed in those terms. Since the number of

productions in this model is low and the number of production cycles is proportional to the number of chunk retrievals, we will focus here on the total number of memory retrievals. That number for one successful inference is roughly equal to the product of the total number of inferences attempted by the average number of retrievals for matching the premise(s) of one inference. These are the numbers along the Y axis for Figures 52 and 53, respectively. The result would be a 3-D plot of the total number of retrievals as a function of log number of inferences and log number of chunks (the X axes in Figures 52 and 53, respectively). Whereas the latter is the simplest measure of problem complexity for evidence marshalling, the number of inferences present in Micro cognition is not specifically a function of the problem but instead of the trade-offs adopted in solving the problem. More interestingly, the total number of inferences increases the complexity of finding the right inference, but actually decreases the difficulty of applying it. If the problem space is more densely covered with inference instances, the degree of generalization required decreases because for any given problem one is likely to already have a closer match among existing inferences. This would allow the mismatch penalty used in partial matching inference premises to be increased, and with it the slope of the matching power law distribution. More specifically, doubling the mismatch penalty requires the number of inference instances to be increased by the same factor of two to the power of number of dimensions in the representation. Assuming three slots per chunk (about right for our representation), this means that doubling the mismatch penalty requires increasing the log of the number of chunks by three.

Figure 54 presents the total number of retrievals required to perform a single successful inference, i.e., our measure of solution complexity, as a function of the log of number of chunks, i.e., our measure of problem complexity. The four curves correspond to values of log number of inference in memory (Figure 52 X axis) of 1 (Series1), 4 (Series2), 7 (Series3) and 10 (Series4). As discussed in the previous paragraph, these correspond to mismatch penalty values of 0.075, 0.15, 0.3 and 0.6, respectively. Figure 54 is basically a combination of Figures 52 and 53 for a given value of the focus slope, 0.1 in this case. Series 1 shows that very few inferences results in the most efficient process for small number of chunks (e.g., under 1000) because the cost of generalization is quite low, but the complexity increases exponentially with the log number of chunks because few inferences requires a very permissive generalization process that results in high matching costs for large number of chunks. At the other extreme, Series 4 shows that large number of inferences make the complexity almost insensitive to the total number of chunks because little generalization is required, but the overall complexity is generally high. Intermediate values of number of inferences show linear (Series2) and logarithmic (Series3) growth of solution complexity as a function of problem complexity, but with increasing starting costs that only become advantageous for problems involving many thousand of chunks. Either of these curves

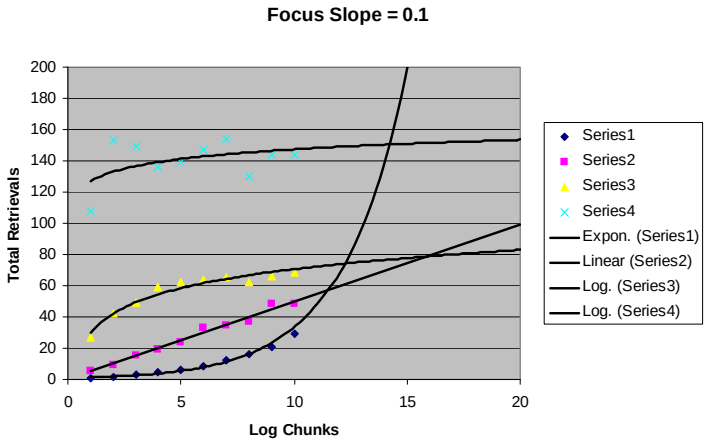


Figure 54. Solution complexity as a function of problem complexity

would probably provide a satisfactory solution, but a combination would be even better. For small problems (log chunks <= 10, i.e., under 1000 chunks), a low number of inferences (Series1) offers very low, stable complexity. As those costs start to climb, adding about an order of magnitude more inferences (Series2) switches to linear growth in solution complexity until about 50,000 chunks (i.e., log <= 15). For problems over that size, adding another order of magnitude more inferences (Series3) provides very tractable logarithmic growth in solution complexity as a function of problem complexity. The composite curve provides a theoretical basis for switching resources between cognitive layer components as a function of problem complexity.

5.1.1.4 Open Issues

This analysis focuses entirely on the Micro cognitive level at the expense of other cognitive levels (Proto and Macro) and other layers, especially hardware. Other cognitive levels will carry their own complexity. For example, increasing the accuracy of Proto clustering might require an inordinate increase in number of computation cycles that would negate the benefits of improved focusing at the Micro level. Similarly, the cost for Macro to generate an order of magnitude more inferences might be so large that the system would be more efficient by tolerating a faster rise in Micro complexity as problem size grows. Determining these tradeoffs requires conducting similar analyses for Proto and Macro Cognition and integrating their costs where parameters for focus slope and total number of inference instances are considered, respectively.

This analysis measures solution complexity in terms of numbers of retrievals, the basic cognitive step (together with production matching) of Micro cognition. However, each memory retrieval is itself a complex process that involves the matching of thousands or even millions of chunks. The actual complexity of that process will depend upon how efficiently the AVM layer will map these basic cognitive steps onto hardware operations. Conversely, each retrieval need not happen sequentially. Multiple inferences could be attempted simultaneously if enough parallelism is available. Therefore, the hardware layer will have to be taken into account before our measure of solution complexity can be related to external dimensions such as time per inference.

5.1.2 Macro Cognition Complexity Analyses

This section outlines the computational complexity of the Macro Cognition component, focusing on first knowledge search and then problem search. As discussed above, problem search is inherently an expensive process (in the worst-case, $O(b^d)$, where b is the average branching factor and d, the average depth of a solution). As we will outline, the specific architectural approach of C3I1 bounds this worst-case expense, with the result that the computational requirements of the deliberate reasoning component in the architecture does not overwhelm the total processing requirements of the complete architecture.

5.1.2.1 Knowledge Search

As discussed above, a key requirement for Macro Cognition is efficient knowledge search and, in Phase I, Soar-as-Macro Cognition met this requirement chiefly via the Rete pattern matcher.

While there are some degenerate cases, Rete is highly optimized for pattern-matching on serial computers, with a worst case computational complexity linear in the size of the knowledge base [Forgy82]. Further, as empirical and theoretical work in the 1980's showed, there is little to be gained from parallelizing or otherwise trying to speed up the Rete matching process [Nayak88]. Because Rete is already highly efficient, we did not focus our effort on improving knowledge search within Macro Cognition in Phase I and assume, in the problem search analysis, that the cost of an individual knowledge retrieval step is O(*n*).

5.1.2.2 Problem Search

The total size of the search spaces of the potential application domains explored in Phase I are infinite and the depth of any particular solution could be very large. Further, within these domains, there are many possible operators, resulting in potentially very large branching factors. These factors suggest that any approach that depends on problem search (i.e., Macro Cognition) alone is *bound to fail* in all but the most trivial of problems. There are three ways in which the overall C3I1 architecture reduces the complexity of individual problems that need to be solved by Macro Cognition: subproblem definition, attentional filtering, and relevance estimation.

Figure 55 illustrates the basic concept. Macro Cognition alone would be presented with an infinite search space and a problem space of significant breadth and potentially large depth. The action of the three mechanisms results in progressively finer focusing of Macro Cognition's activity.

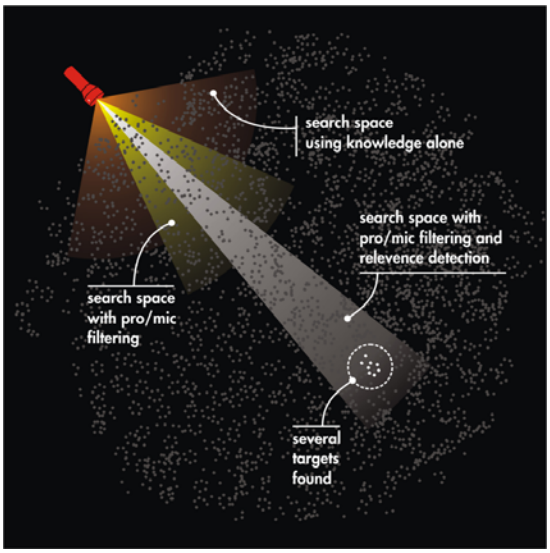


Figure 55. Conceptual illustration of the focusing that helps bound the computational complexity of Macro Cognition problem search.

Subproblem definition: Instead of being presented with large, open-ended problems (e.g., plan a UAV mission, determine if there's a terrorist threat in these intel reports), Macro Cognition is tasked with much less open-ended, more defined problems (find a good route among these five points; find a connection between these two facts).

Attentional filtering: In addition to defining a subproblem, Proto cognition and Micro cognition also define specific items of interest. In Phase I, we explored a number of mechanisms for defining the attentional focus, including both explicit definition ("attend to these items") and implicit definition, such as the level of activation in memory at the time of a request. The effect of this mechanism is to further focus the "beam" of the Macro cognitive search, allowing Macro Cognition to ignore (or at least discount) memories outside of the attentional focus. The result of these two mechanisms is that rather than having to solve a broad problem, Micro cognition "points" Macro Cognition in a good direction for solving a much more narrowly conceived problem.

Relevance estimation: Subproblem definition and attentional filtering are largely static filters. Thus, they have some initial effect on reducing the depth of a solution (presumably, smaller subproblems are found in relatively few steps in comparison to the overall problem) and bounding of branching factor. Within Macro Cognition, the relevance estimation function, as described above, provides *dynamic* bounding of the search space. In the ideal case, perfect relevance estimation would lead directly to a solution in a number of steps equivalent to the depth of a solution. That is, at each step in the problem search, relevance estimation would identify the step that leads to the (best) solution. Obviously, perfect relevance estimation is not possible. In the worst case, assuming the relevance estimation process itself is not as costly as problem search, relevance estimation would not change the worst-case complexity of the search, $O(b^d)$.

We performed a simple theoretical analysis to show the relative benefits of subproblem definition, attentional filtering, and relevance estimation. From the standpoint of Macro Cognition, the search required to answer any query to the cognitive layer will be exponential. In general, a heuristic and filter-free search would be through a graph with branching factor, b, with the target at depth d, yielding bd potential targets. Figure 56 shows a balanced tree of depth $d = 3$ and branching factor $b = 3$ with 27 potential target nodes. While it is likely that the overall size of an actual problem will be much larger, this small tree highlights the potential impact of subproblem definition, where small depths of solution and smaller branching factors are expected.

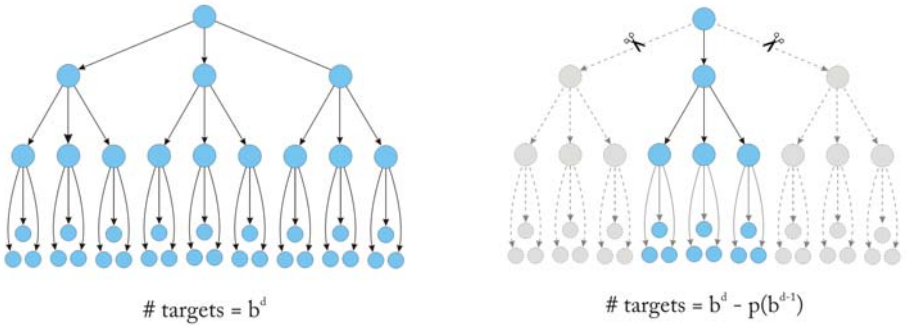


Figure 56. Simple illustration of the size of a search space (left) and the effect of attentional filtering (right).

The effect of the attentional filtering and attention focusing power provided by Proto Cognition and Micro Cognition can be viewed as a pruning of branches from the *root* of the search tree. In other words, filtering reduces the branching factor at the root, but the branching factor beyond is unchanged. Figure 52 illustrates that Proto/Micro filtering prunes two branches ($p = 2$) from the root; the branching factor within the remaining subtree/s are unaffected. The potential targets have been reduced from 27 to 9.

The pruning via Proto and Micro filtering has reduced the search space for this problem ($b = 3$, $d = 3$). However, with larger branching factors and deeper search depths, the pruned search space can still be quite large. To reduce the space further, Macro Cognition will use a relevance estimation heuristic. By evaluating the potential relevance of all options of an expanded node,

Macro Cognition can evaluate which nodes may be more profitable to explore than others. Nodes whose relevance estimation is below a threshold are not expanded. If all options of an expanded node are below the threshold, then a sibling node is explored. The effect of relevance estimation on the size of the search space is to reduce the branching factor of the entire subtree, which can produce very large reductions in the search space. Figure 57 shows the situation where relevance detection prunes one-third of the branches and explore two-thirds (*rd* = 0.67). The potential targets have been reduced from 9 to 4.

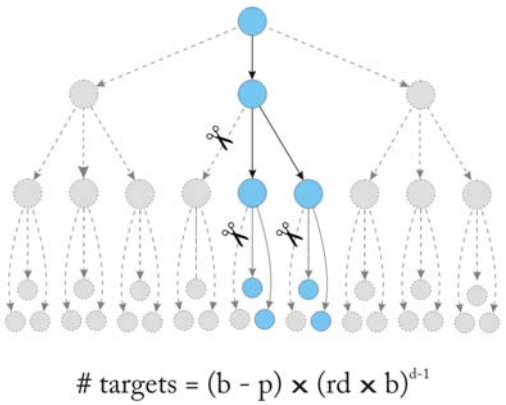


Figure 57. Effect of dynamic relevance estimation on the overall search space, assuming 33% of notes are filtered.

The reductions in search space are meager for this example problem with $b = 3$; $d = 3$, $p = 2$, $rd = 0.67$. However, as target depth increases, the overall effect of relevance estimation can be significant, as shown in Figure 58. In this diagram, the x-axis represents the number of problem search steps (*d*) and the y-axis is the log of potential targets. While filtering has an effect on the complexity of the search, that effect is relatively modest because it only reduces the size of the initial problem. With relevance estimation, the size of the benefit becomes more pronounced as the depth of the search increases, because relevance estimation is actively pruning the (effective) search tree at each step in the search. As a result, for example at *d*=9, almost two orders of magnitude fewer nodes need to be examined than either the problem with no filtering or with attentional filtering.

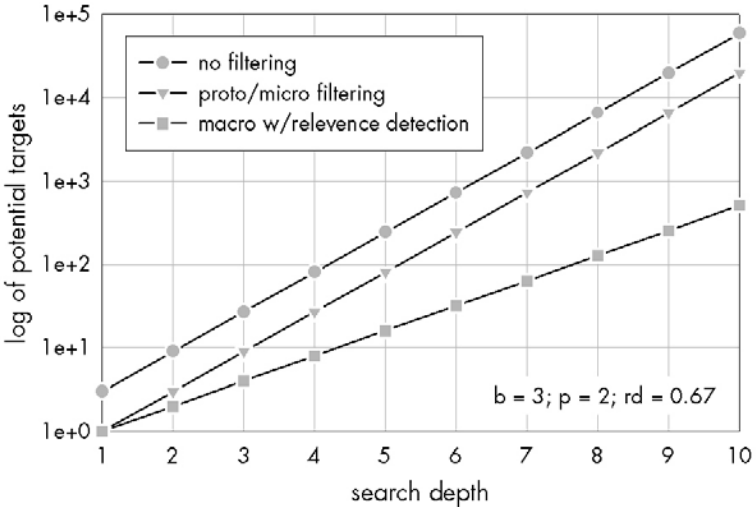


Figure 58. Search depth vs. log of potential targets for hypothetical problem in which branching factor =3, filtering effect is 2, and relevance estimation removes 33% of available options.

Figure 59 shows a similar plot for both a more realistic branching factor and greater effectiveness of relevance estimation ($b = 8$; $p = 2$, $rd = 0.25$). Although there are four orders of magnitude more potential targets in this problem than for $b = 3$, Macro Cognition accompanied by effective relevance estimation effective bound the problem. Another way of interpreting the diagram is that for a fixed number of target explorations (e.g., $n = 100$), without any filtering, Macro Cognition would only find solutions at a depth of 2, while filtering would increase the effective depth to approximately 2.5 while this very effectively relevance estimation could extend the depth of the search to 9.

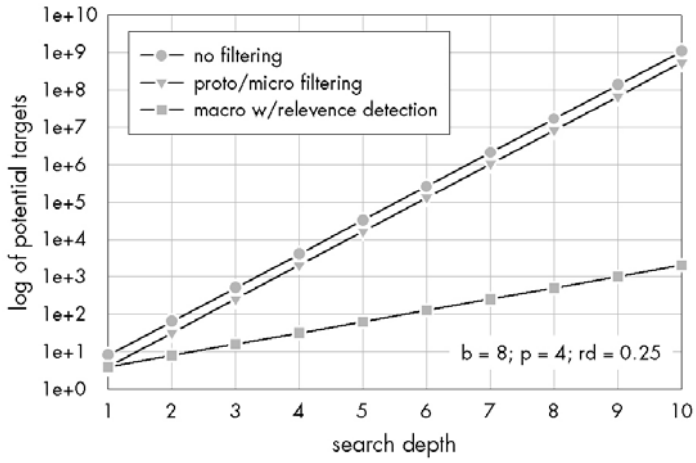


Figure 59. Search depth vs. log of potential targets for hypothetical problem in which branching factor =8, filtering effect is 4, and relevance estimation removes 75% of available options.

Obviously, this is only a theoretical analysis and finding effective solutions to the relevance estimation problem is an open question. The overall impact of this analysis is to show that while filtering is helpful, subproblem definition is the most effective available means for reducing the complexity of Macro Cognition's overall search and relevance estimation, as appropriate techniques become available, will allow a deeper search of the problem space in a given unit of time.

5.1.3 Proto Cognition Complexity Analyses

The best-case computational complexity of Proto cognition is O(1). However, this analysis assumes the following:

- Fully scalable (O(m)) hardware (where m is the number of processors)
- Massive numbers of processors available
- No (or, very low) cost AVL mapping of processes to processors

In a more realistic case, the computational complexity will be O(n/α), since not all assumptions will be completely satisfied.

5.1.3.1 SODAS Complexity Analysis

Hierarchical clustering algorithms produce a dendrogram from an unordered set of data. Classical agglomerative clustering algorithms have a time complexity of O(n^2), where n is the number of data items [Clark95] (Figure 60). Furthermore, agglomerative clustering methods must re-cluster the entire dataset from scratch if data is inserted or deleted, again taking O(n^2) time, while SODAS can integrate the changes in at most O(n log(n)) time. Agglomerative clustering can be parallelized with a time complexity of O(n), but only using specialized parallel hardware with a global memory. If a parallel architecture with local memory is used, then agglomerative clustering has a time complexity of O(n log(n)), but this is no better than the serial version of SODAS. SODAS itself can be parallelized using local memory with a time complexity of O(log(n))

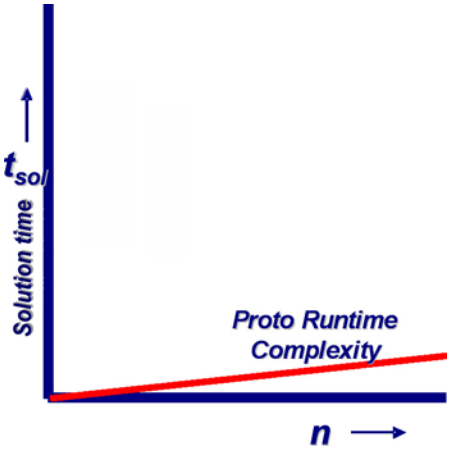


Figure 60. Proto cognition's realistic runtime complexity is O(n/α)

5.1.3.1.1 Classical agglomerative clustering

5.1.3.1.1.1 Static Clustering Time Complexity

The time complexity of classical agglomerative clustering is at best $O(n^2)$. (Naive implementations are $O(n^3)$. If a linkage method other than single linkage is used, then the complexity becomes $O(n^2 \log(n))$.) This is due to the fact that the algorithm must compute the similarity of each pair of documents (see [Olson95] for a full analysis).

5.1.3.1.1.2 Dynamic Clustering Time Complexity

Unlike SODAS, classical agglomerative clustering has no promote mechanism. Hence, it has no way to undo a clustering decision if a new document is inserted that makes another clustering more optimal. Furthermore, as more and more documents are inserted or deleted, the initial clustering will become less and less relevant. The only way to guarantee an optimal clustering is to redo the entire clustering after each insertion or deletion. Thus, the time to insert or delete documents is again $O(n^2)$.

5.1.3.1.1.3 Parallel Time Complexity

$O(n)$ algorithms exist for specialized parallel hardware (having a common Common Real Common Write (CRCW) memory), assuming one processor per document [Olson95]. However, these algorithms will not work on parallel architectures with local memory, such as a LAN of PCs. For architectures with local memory, Olson gives algorithms with $O(n \log(n))$ time complexity.

5.1.3.1.2 SODAS

SODAS consists of a clustering algorithm and a foraging algorithm. The foraging algorithm can be used with a cluster tree produced by any mechanism. The search time on a balanced tree is $O(\log(n))$; foraging prunes the tree, providing the same effect (for given n) as increasing the branching factor, thus reducing the search time by a constant factor. This analysis focuses only on SODAS' clustering algorithm, not the foraging algorithm. SODAS clustering makes use of two fundamental operations: merge and promote, along with a homogeneity metric. In the merge operation, two child clusters of the active cluster are merged together if they are more similar than the cluster as a whole; the new merged cluster becomes a child of the active cluster. The promote operation promotes a child cluster to be a sibling of the active node if it is an outlier within the active cluster. The homogeneity metric is used to calculate the homogeneity of a cluster, similarity of two clusters, or the similarity of two documents. For full details, see [Parunak06].

5.1.3.1.2.1 Assumptions

The SODAS time complexity analysis makes the following assumptions:

- The dimensionality of the document feature vector $d << n$. This is a reasonable assumption for the massive datasets that SODAS is designed for (e.g., millions of documents and thousands of features).
- For all clusters, the cluster size (i.e., the number of child nodes) $c << n$. SODAS can be initialized with a random tree that has $c < 10$, and the promote and merge operators can be modified to only consider a small sampling of a cluster's children. In this way, c can be effectively kept small.
- The cluster tree remains balanced, so that the maximum depth of the tree is O(log(n)). This holds true in practice.

5.1.3.1.2.2 Homogeneity Metric Runtime Complexity

The homogeneity metric (see Section 4 of [Parunak06]) is defined over a set of clusters or documents. To calculate the similarity of two clusters or two documents, simply form a cluster comprised of those clusters or documents, and apply the metric to the resulting cluster. Its time complexity is based on the cluster size c and the dimensionality of the feature vector d, since the calculations must loop over both the subclusters and the features:

HomogeneityTime = O(c d)

If $c << n$ and $d << n$, as assumed, then this reduces to O(1); i.e., it is a constant-time operation wrt n.

5.1.3.1.2.3 Merge Operation Runtime Complexity

In the merge operation, two child clusters of the active cluster are merged together if their combined homogeneity is greater than that of the cluster as a whole; the new merged cluster becomes a child of the active cluster. The number of candidate children for merger is c_m. If the cluster size c is too large, c_m can be a sampled subset of the node's children. The number of candidate pairs for merger is:

pairs = (c_m □ (c_m − 1))/2

The time complexity to compare all pairs' homogeneity with the current cluster's is:

MergeTime= O(pairs □ HomogeneityTime) = O(c_m (c_m – 1) / 2)

If $c_m <= c << n$ and $d << n$, as assumed, then this reduces to O(1).

5.1.3.1.3 Promote Operation Runtime Complexity

The promote operation promotes a child cluster to be a sibling of the active node if it is an outlier within the active cluster, i.e., if the homogeneity of the cluster without the child is much greater than the cluster's current homogeneity. The number of candidate children for promotion is c_p. If

the cluster size c is too large, c_p can be a sampled subset of the node's children. The time to examine all promotion candidates for a given cluster is:

PromoteTime = O(c_p □ HomogeneityTime)

If c_p <= c << n and d << n, as assumed, then this reduces to O(1).

5.1.3.1.3.1 SODAS Static Runtime Complexity

Assume an initial random cluster tree. In the worst case, SODAS must promote documents to the root and then merge them back down to their correct position. Since the tree has a depth of O(log(n)), it requires O(log(n)) promote and merge operations to accomplish this. Due to stochasticity, there may be erroneous promote or merge operations, which will require more operations to undo the incorrect move; however, this simply adds a factor of 2m log(n), where m is the number of erroneous promote and merge operations:

SODASRunTime = O(n □ (log(n)+ 2 m □ log(n))□ PromoteTime)+ O(n □ (log(n)+ 2 m □ log(n))□ MergeTime)

Disregarding constants, this reduces to:

SODASRunTime = O(n □ log(n))

5.1.3.1.3.2 SODAS Dynamic Runtime Complexity

In the worst case, SODAS must recluster the entire tree after a document is inserted or deleted, which is O(n * log(n)). However, SODAS' promote operation allows it to fix a clustering without redoing the entire structure, so this will likely be reduced by a large constant factor in practice.

5.1.3.1.3.3 Impact of Parallelizing SODAS

All SODAS operations use only information from the active node and its local neighborhood. Thus, the only constraint on parallelization is the necessity of maintaining the consistency of the cluster tree. If there are p threads and n processors that can run simultaneously, p <= 1, then SODAS' runtime complexity reduces to:

ParallelSODASRunTime = O(SODASRunTime O(p n)) = O(log(n)/p)

p reflects two influences: the number of nodes in the hierarchy assigned to each processor (thus the total number of processors), and the avoidance of conflicts among processors. For example, updating a single node in the hierarchy may modify c children and a single parent, and if these happen to reside on different processors than the active node, those processors should not be executing simultaneously.

How likely is such a conflict? Let's compute the number of internal nodes in the hierarchy for branching factor k and n documents. Each level contributes k_i nodes, starting with $i = 0$ (the root), and there are $\log_k(n)$ levels including the leaves, or $\log_k(n)$ - 1 internal levels. This sum is:

$$\sum_{i=0}^{\log_k n} k^i ,$$

which reduces to

$$(n - 1) / (k - 1)$$

Assign j nodes in the hierarchy to each processor, so that we have $(n\text{-}1)/(j(k\text{-}1))$ processors in all. Activating a single node will generate conflicts with $c + 1$ nodes, which may or may not be on the same processor with the active node. The clustering process will tend to keep adjacent nodes on the same processor, but in the worst case each affected node might be on a different processor, and on average $(c+1)/j$ processors will be affected. Thus the proportion of processors that will be disabled by a given activation is $(c+1)\ (k\text{-}1)/(n\text{-}1)$. If $c << n$, as assumed, the probability of conflict is vanishingly small. Since j is a constant, we have:

ParallelSODASRunTime = O(log(n))

5.1.3.1.4 SODAS Complexity Analysis Summary

This analysis has shown that SODAS' clustering process has a serial time complexity of O(n log(n)) and a parallel time complexity of O(log(n)). Furthermore, its dynamic clustering time complexity is at worst O(n log(n)).

5.2 Implementation Considerations

The architecture of cognition is a continuing research program and the attempt to apply the general methods to this specific application should further extend our understanding of these frameworks. Experimentation and analysis show that specialized hardware can significantly outperform general purpose processing (in some cases by orders of magnitude) for kernels applied to the C3I1 framework. However, along with Amdahl's law, there are additional obstacles to achieving system level performance on par with the improvements of the individual critical components. For example, the transformation of sensory data into a symmetry-invariant form suitable for matching and retrieval in a limited memory can be resource intensive and difficult to generalize.

In order to help system designers select cognitive processing implementation options, the designers must estimate how key application kernels perform over prototypical framework implementation options. It is important that the application kernels and implementation emulations are representative of the given application and target processing architecture, respectively. An example of an indicator of performance identified as important for C3I1 is "chunks-per-second," but in general, measures of performance of a cognitive system may not be so clear cut, and may not apply to different frameworks.

The continued top-to-bottom implementation of applications using cognitive techniques should yield additional insights and capabilities. For example, while we have begun to study the kernels and cognitive frameworks in isolation, integrated applications should provide insights and opportunities, e.g., for specialized operators working within application-compatible reduced reliability or exploiting temporal or spatial locality in the control and data flow across kernels. The cognitive frameworks are, in a sense, interpreters running a high-level cognitive program; with more insight into the structure and behavior of this program we expect to be able to develop specialized system and hardware mechanisms to exploit them. System and hardware support that reduces the cost of using cognitive frameworks should facilitate further application of cognitive techniques, increasing the intelligence of important embedded systems and large scale ground-based systems.

5.2.1 Macro Cognition

The Proto Cognitive and Micro Cognitive layers are the primary beneficiaries of specialization in the hardware implementation. Special hashing hardware and efficient shared memory architectures, however, benefit macro-cognitive processing. Concurrency in a single instance of a Rete graph used in the core of Macro Cognition is limited to about 10X. Spending hardware and software resources to leverage this concurrency has little payoff relative to similar efforts in micro and proto-cognitive layers.

5.2.2 Micro Cognition

The primary matching operations of Micro Cognition are parallelizable. Multi-core processing presents a direct per-chip gain. In addition, the specialization of memory with configurable logic presents greater than 100X improvement in the matching performance indicator "chunks-per-second," which is the number of data vectors in memory that can be compared with an input operand according to a configurable matching operator.

5.2.3 Proto Cognition

Proto Cognition is implemented directly in asynchronous logic and on-chip processors for dramatic increase in performance (greater than 200X) over a conventional processor of comparable area. The productivity issues associated with leveraging the tremendous gains are being addressed with domain-specific generators, and the programming abstractions provided by AVM/L.

5.3 Proposed Follow-Up Research

In terms of the methodological hypothesis, our explorations identified potential opportunities for much tighter, more synergistic integrations of the components. Integration between swarming and ACT-R will focus on unifying the ACT-R activation calculus and the swarming output, including devising methods for learning within ACT-R to influence Swarming. Integration between ACT-R and Soar based on the pattern of usage in the Prototypes would center on impasses and chunking. ACT-R currently calls upon Soar when no relevant knowledge exists to

apply directly, which is a concept very similar to the Soar impasse mechanism, and, as well, not a process directly supported in ACT-R. The information returned by Soar is a summary of the result of its reasoning. Thus, this process is functionally identical to Soar's chunking and makes it a good candidate for the focus of integration for Soar and ACT-R.

As mentioned, other alternatives are possible for the cognitions of C3I1. For example, neurologically-inspired clustering might be appropriate for perceptually-dominated domains. These observations point to the question of whether C3I1 is an architecture, making specific commitments to specific mechanisms, or a framework, representing a design pattern for building general, intelligent systems but not making specific commitments to representations and algorithms. While our intention is to follow up the promise observed in the Prototypes by investigating more fine-grained integration of the current technologies, it would also be equally valid to explore and evaluate alternative technologies. Similarly, the C3I1 decomposition also serves as a suggestive guide for developing less brittle and more scalable solutions to specific classes of domain problems (such as route planning), where the goal is a point solution. All of these options point to the value of considering existing cognitive architectures as computational primitives for integration into larger and increasingly capable intelligent systems.

6. References

[Agha85] Agha, G. Actors: A model of concurrent computation in distributed systems. MIT AI Lab. technical report 844, 1985.

[Altmann99] Altmann, E.M. and B.E. John. Episodic Indexing: A model of memory for attention events. Cognitive Science, 23(2), 117-156,1999.

[Amduka06] Amduka, M., J.C. Russo, K. Pederson, R. Lethin, J. Springer, R. Manohar, and R. Melhem. Enabling Cognitive Architectures for UAV Mission Planning. *In Proc. of the Tenth Annual High Performance Embedded Computing Workshop, 2006.*

[Anderson98] Anderson, J.R. and C. Lebiere. The Atomic Components of Thought. Erl-baum, Mahwah, NJ, 1998.

[Anderson91] Anderson, J.R. and L.J. Schooler, L. J. Reflections of the environment in memory. Psychological Science, Vol. 2, pp. 396-408, 1991.

[Anderson03] Anderson, J.R. and C. Lebiere. The Newell test for a theory of cognition. Behavioral and Brain Science, Vol. 26, pp. 587-637, 2003.

[Anderson05] Anderson, T., D.A Schum, and W. Twining. Analysis of Evidence, Cambridge University Press, 2005.

[Applegate98] Applegate, D., R. Bixby, V. Chvatal, and W. Cook. On the solution of Traveling Salesman Problem. International Congress of Mathematicians, 1998.

[Ball96] Ball, P. The Self-Made Tapestry: Pattern Formation in Nature. Princeton, NJ, Princeton University Press, 1996.

[Beckett96] Beckett, D.J. and P.H. Welch. A Strict occam Design Tool. In Proceedings of UK Parallel, edited by C.R.Jesshope et al., pp. 53-69, Springer-Verlag, ISBN 3-540-76068-7, 1996.

[Boehm05] Boehm, H.-J. Threads cannot be implemented as a library. In PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA, 2005), ACM Press, pp. 261-268.

[Brueckner00] Brueckner, Sven. Return from the Ant: Synthetic Ecosystems for Manufacturing Control. Dr.rer.nat. Thesis at Humboldt University Berlin, Department of Computer Science. http://dochost.rz.hu-berlin.de/dissertationen/brueckner-sven-2000-06-21/PDF/Brueckner.pdf, 2000.

[Brueckner02] Brueckner, Sven A. and H. Van Dyke Parunak. Swarming Agents for Distributed Pattern Detection and Classification. In Proceedings of Workshop on Ubiquitous Computing,

AAMAS 2002, Bologna, Italy, 2002.
http://www.newvectors.net/staff/parunakv/PatternDetection01.pdf.

[Brueckner03] Brueckner, Sven A. and H. Van Dyke Parunak. Resource-aware exploration of emergent dynamics of simulated systems. In Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 2002), pp. 781-788. 2003.
http://www.newvectors.net/staff/parunakv/AAMAS03APSE.pdf

[Chong05] Chong,R. S. and R.E. Wray. Constraints on Architectural Models: Elements of ACT-R, Soar and EPIC in Human Learning and Performance. In K. Gluck & R. Pew (Eds.), Modeling Human Behavior with Integrated Cognitive Architectures: Comparison, Evaluation, and Validation (pp. 237-304): Lawrence Erlbaum Associates, 2005.

[Cho07] Cho, S., J.R. Martin, R. Xu, M.H. Hammoud, and R. Melhem. CA-RAM: A High-Performance Memory Substrate for Search-Intensive Applications, *In Proc. of the IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS 2007)*, pp. 230-241, 2007.

[Coarfa05] Coarfa,C., Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarria-Miranda. An evaluation of global address space languages: CoArray Fortran and Unified Parallel C. In PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (New York, NY, USA, 2005), ACM Press, pp. 36-47.

[Dean04] Dean, J. and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In 6th Symposium on Operating System Design and Implementation (OSDI '04), 2004.

[DeHon05] DeHon, A. and M. deLorimier. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 75-85, February 2005.

[Dietterich86] Dietterich, T.G. Learning at the knowledge level. Machine Learning, 1, 287-315, 1986.

[Doorenbos94] Doorenbos, R.B. Combining left and right unlinking for matching a large number of learned rules. Paper presented at the Twelfth National Conference on Artificial Intelligence (AAAI-94), Seattle, Washington, 1994.

[Doorenbos95] Doorenbos, R.B. *Production Matching for Large Learning Systems,* Ph.D. Thesis, CMU, 1995.

[Doyle79] Doyle, J. A truth maintenance system. Artificial Intelligence, 12, 231-272, 1979

[Epstein06] Epstein, S.L. In Support of Pragmatic Computation. In C. Lebiere & R. E. Wray (Eds.), Proceedings of the AAAI Spring Symposium on Cognitive Science Meets AI-hard Problems. Stanford: AAAI Press, 2006.

[Epstein04] Epstein, S.L. and T. Ligorio. Fast and Frugal Reasoning Enhances a Solver for Really Hard Problems. In Proceedings of the Cognitive Science Chicago, 2004.

[Forgy82] Forgy, C. Rete: A Fast Algorithm for the Many Pattern/Many Object Pat-tern Match Problem. In Artificial Intelligence, 19, pp 17-37, 1982.

[Forgy79] Forgy, C. On the efficient implementation of production systems. Ph.D. Thesis, Carnegie-Mellon University, 1979.

[Frigo98] Frigo, M., C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. In PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (New York, NY, USA, 1998), ACM Press, pp. 212-223.

[Gigerenzer99] Gigerenzer, G., P.M. Todd, and ABC Research Group. Simple heuristics that make us smart. NYC: Oxford University Press, 1999.

[GraphStep06] deLorimier, M., N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T.E. Uribe, T.F. Knight, Jr., and A. DeHon. GraphStep: A System Architecture for Sparce-Graph Algorithms. In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, pages 243-151. IEEE, 2006.

[Gordon96] Gordon, A.D. Hierarchical Classification. In P. Arabie, L.J. Hubert, and G. DeSoete, Editors, Clustering and Classification, pages 65-121. World Scientific, River Edge, NJ, 1996.

[Grasse59] Grasse, P.P. La Reconstruction du nid et les Coordinations Inter-Individuelles chez Bel-licositermes Natalensis et Cubitermes sp. La théorie de la Stigmergie: Essai d'interpréta-tion du Comportement des Termites Constructeurs. Insectes Sociaux, 6:41-84, 1959.

[Gupta89] Gupta, A., C. Forgy, and A. Newell. High Speed Implementations of Rule-Based Systems. In Transactions on Computer Systems 7(2), May 1989.

[Hanks93] Hanks, S., Pollack,M.E., & Cohen,P.R. (1993). Benchmarks, Test Beds, Controlled Experimentation, and the Design of Agent Architectures. AI Magazine, 14, 17-42.

[Harris05] Harris, T., S.Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In ACM Conference on Principles and Practice of Parallel Programming. ACM Press, 2005.

[Hoare78] Hoare, C.A.R. Communicating sequential processes. In Communications of the ACM, 21(8), 1978, pp. 666-677.

[Huffman95] Huffman, S.B. and J.E. Laird. Flexibly instructable agents. Journal of Artificial Intelligence Research, 3, 271-324, 1995.

[Jain99] Jain, A.K., M.N. Murty, and P.J. Flynn. Data Clustering: A Review. ACM Computing Surveys 31(3), 1999.

[Jones99] Jones, R.M., J.E. Laird, P.E. Nielsen, K.J. Coulter, P.G. Kenny, and F.V. Koss. Automated Intelligent Pilots for Combat Flight Simulation. AI Magazine, 20(1), 27-42, 1999.

[Kale93] Kale, L.V. and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In OOPSLA '93: Proceedings of the eighth annual conference on object-oriented programming systems, languages, and applications (New York, NY, USA, 1993), ACM Press, pp. 91-108.

[Laird87] Laird, J.E., A. Newell, and P.S. Rosenbloom. Soar: An architecture for general intelligence. Artificial Intelligence, 33(3), 1-64, 1987.

[Laird91] Laird, J.E., E.S. Yager, M. Hucka, and M. Tuck. Robo-Soar: An integration of external interaction, planning and learning using Soar. Robotics and Autonomous Systems, 8(1-2), 113-129, 1991.

[Laird95] Laird, J.E. and P.S. Rosenbloom. The evolution of the Soar cognitive architecture. In D. Steir & T. Mitchell (Eds.), Mind Matters. Hillsdale, NJ: Lawrence Erlbaum Associates., 1995.

[Laird96] Laird, J. and P. Rosenbloom. The evolution of the Soar cognitive architecture. In D.M. Steier & T.M. Mitchell (Eds.), Mind Matters: A tribute to Allen Newell, pp. 1-50. Erlbaum, Mahwah, NJ, 1996.

[Laird96a] Laird, J.E., D.J. Pearson, R.M. Jones, and R.E. Wray. Dynamic knowledge integration during plan execution. In Proceedings of the Papers from the 1996 AAAI Fall Symposium on Plan Execution: Problems and Issues, Cambridge, MA. 92-98,1996.

[Lea00] Lea, D. Concurrent Programming in Java: Design Principles and Pattern, 2nd Edition. Prentice Hall, 2000.

[Lebiere03] Lebiere, C., R. Gray, D. Salvucci, and R. West. Choice and Learning under Uncertainty: A Case Study in Baseball Batting. In Proceedings of the 25th Annual Meeting of the Cognitive Science Society. pp. 704-709, 2003.

[Lehman95] Lehman, J.F., J.V. Dyke, R. Rubinoff. Natural language processing for Intelligent Forces (IFORs): Comprehension and Generation in the Air Combat Domain. Paper presented

at the Fifth Conference on Computer Generated Forces and Behavioral Representation, Orlando, FL, May 1995.

[Lehman98] Lehman, J.F., R.L. Lewis, and A. Newell. Architectural influences on language comprehension. In Z. Pylyshyn (Ed.), Cognitive Architecture. Norwood, NJ: Ablex, 1998.

[MPI98] MPI http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, 1998.

[Manohar06] Manohar, R. Reconfigurable Asynchronous Logic. Invited paper, Proceedings of the IEEE Custom Integrated Circuits Conference, 2006.

[Manohar04] Manohar, R. and K.M. Chandy. Delta-Dataflow Networks for Event Stream Processing. Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, November 2004.

[Miller96] Miller, C.S. and J.E. Laird. Accounting for graded performance within a discrete search framework. Cognitive Science, 20, 499-537, 1996.

[Mitchell97] Mitchell, T.M. Machine Learning. McGraw Hill, 1997.

[Nayak88] Nayak, P., A. Gupta, and P. Rosenbloom. Comparison of the Rete and Treat production matchers for SOAR (a summary). In Proceedings of the National Conference on Artificial Intelligence, 693-698, August 1988.

[Newell80] Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), Attention and Performance VIII. Hillsdale, NJ: Erlbaum, 1980.

[Newell90] Newell, A. Unified Theories of Cognition. Cambridge, Massachusetts: Harvard University Press, 1990.

[Newell00] Newell, A. Unified Theories of Cognition, Harvard University Press, 2000.

[Newell91] Newell, A., G.R. Yost, J.E. Laird, P.S. Rosenbloom, and E.M. Altmann. Formulating the Problem Space Computational Model. In R. F. Rashid (Ed.), CMU Computer Science: A 25th Anniversary Commemorative, pp. 255-293, 1991. ACM Press/Addison-Wesley.

[Nielsen02] Nielsen, P., J. Beard, J. Kiessel, and J. Beisaw. Robust Behavior Modeling. In Proceedings of the 11th Computer Generated Forces Conference, 2002.

[Nuxoll04] Nuxoll, A. and J.E. Laird. A Cognitive Model of Episodic Memory Integrated with a General Cognitive Architecture. Paper presented at the International Conference on Cognitive Modeling, Pittsburgh, PA, 2004.

[Odell03] Odell, J.J., H.V.D. Parunak, S. Brueckner, and J. Sauter. Temporal Aspects of Dynamic Role Assignment. In Proceedings of Workshop on Agent-Oriented Software Engineering (AOSE03) at AAMAS03, Melbourne, AU, Springer, 2003. http://www.newvectors.net/staff/parunakv/TemporalAspects03.pdf

[Olson95] Olson, Clark F. Parallel algorithms for hierarchical clustering. Parallel Computing 21(8): 1313-1325, 1995. http://citeseer.ist.psu.edu/article/olson93parallel.html

[Parunak04] Parunak, H.V.D., S.A. Brueckner, and J. Sauter. Digital Pheromones for Coordination of Unmanned Vehicles, In Proceedings of Workshop on Environments for Multi-Agent Systems (E4MAS 2004), Springer, 2004.

[Parunak02] Parunak, H.V.D., M. Purcell, R. O'Connell. Digital Pheromones for Autonomous Coordination of Swarming UAV's. In Proceedings of First AIAA Unmanned Aerospace Vehicles, Systems, Technologies, and Operations Conference, Norfolk, VA, AIAA, 2002. http://www.newvectors.net/staff/parunakv/AIAA02.pdf.

[Parunak06] Parunak, H.V.D., R. Rohwer, T.C. Belding, and S.A. Brueckner. Dynamic Decentralized Any-Time Hierarchical Clustering. In Proceedings of Proceedings of the Fourth International Workshop on Engineering Self-Organizing Systems (ESOA'06), Hakodate, Japan, Springer S.A., 2006. http://www.newvectors.net/staff/parunakv/SODAS06.pdf

[PCAA-RecAP-TR] Amduka, M., G. Viomentes, M. Craven, M., T. Vu. Reconfigureable Architectures for Perception (RecAP) technical report, DARPA ACIP program.

[Pearson98] Pearson, D.J. and J.E. Laird. Toward incremental knowledge correction for agents in complex environments. Machine Intelligence, 15, 1998.

[Pearson93] Pearson, D.J., S.B. Huffman, M.B. Willis, J.E. Laird, and R.M. Jones. A Symbolic Solution to Intelligent Real-Time Control. Robotics and Autonomous Systems, 11, 279-291, 1993.

[Pfeffer01] Pfeffer, A. IBAL: An Integrated Bayesian Agent Language. Joint Conference on Artificial Intelligence (IJCAI), 2001.

[Ramsey02] Ramsey, N. and A. Pfeffer. Stochastic Lambda Calculus and Monads of Probability, POPL, 2002.

[Rinard98] Rinard, M.C. and M.S. Lam. The design, implementation, and evaluation of Jade. In ACM Trans. on Programming Languages and Systems, 20(3), 1998, pp. 483-545.

[Rosenbloom85] Rosenbloom, P.S., J.E. Laird, J. McDermott, A. Newell, E. Orciuch. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. IEEE Transactions on Pattern Analysis and Machine Intelligence, 7, 561-569, 1985.

[Russell95] Russell, S. and P. Norvig. Artificial Intelligence: A Modern Approach. Upper Saddle River, NJ: Prentice-Hall, 1995.

[Simon81] Simon, H.A. Sciences of the Artificial (2nd ed.). Cambridge, MA: MIT Press, 1981.

[Tambe90] Tambe, M., A. Newell, and P.S. Rosenbloom. The problem of expensive chunks and its solution by restricting expressiveness. Machine Learning, 5, 299-348, 1990.

[Teifel04] Teifel, J. and R. Manohar. Highly Pipelined Asynchronous FPGAs. 12th ACM International Symposium on Field-Programmable Gate Arrays, Monterey, CA, February 2004.

[Wallach00] Wallach, D. and C. Lebiere. Learning of event sequences: An architectural approach. In N. Taatgen (Ed.). In Proceedings of the Third International Conference on Cognitive Modeling, pp. 271-279. Groningen: Universal Press, 2000.

[Washington93] Washington, R. and P.S. Rosenbloom. Applying Problem Solving and Learning to Diagnosis. In P.S. Rosenbloom & J.E. Laird & A. Newell (Eds.), The Soar Papers: Research on Integrated Intelligence, Vol. 1, pp. 674-687, Cambridge, MA: MIT Press, 1993.

[Wolfson97] Wolfson H J. and I. Rigoutsos. Geometric Hashing: An Overview. IEEE Computational Science and Engineering, Vol. 4, No. 4, pp. 10-21, 1977.

[Wray03] Wray, R.E. and J.E. Laird. An architectural approach to consistency in hierarchical execution. Journal of Artificial Intelligence Research, Vol. 19, pp. 355-398, 2003.

[Wray04] Wray, R.E., J.E. Laird, A. Nuxoll, D. Stokes, and A. Kerfoot. Synthetic Adversaries for Urban Combat Training. Paper presented at the 2004 Innovative Applications of Artificial Intelligence Conference, San Jose, CA, 2004.

[Wray04] Wray, R.E., S. Lisse, and J. Beard. Investigating Ontology Infrastructures for Execution-oriented Autonomous Agents. Robotics and Autonomous Systems, Vol. 49(1-2), pp. 113-122, 2004.

[Xu05] Xu, Y., T.K. Ralphs, L. Ladanyi, and M.J. Saltzman. ALPS: A Framework for Implementing Parallel Search Algorithms. In Proceedings of the Ninth INFORMS Computing Society Conference, 2005, p. 319.

[Yokote87] Yokote, Y. and M. Tokoro. Experience and evolution of concurrent Smalltalk. In Conference Proceedings on Object Oriented Programming Systems Languages and Applications, 1987, ACM Press, pp. 406-415.

[Yokote86] Yokote, Y. and M. Tokoro. The design and implementation of Concurrent Smalltalk. In Conference Proceedings on Object Oriented Programming Systems Languages and Applications, 1986, ACM Press, pp. 331-340.

[Young99] Young, R. and R.L. Lewis. The Soar Cognitive Architecture and Human Working Memory. In A. Miyake & P. Shah (Eds.), Models of Working Memory: Mechanisms of Active Maintenance and Executive Control, pp. 224-256, Cambridge, UK: Cambridge University Press, 1999.

7. Appendix

7.1 Protocols and Specification Languages

7.1.1 Cognitive Markup Language (CML)

The knowledge representation for the Cognitive Layer of PCAA is called the Cognitive Markup Language (CML). This was originally defined to be the language by with the Application layer communicates with the Cognition layer, though it evolved to become the internal language of the Cognitive Layer.

CML evolved through roughly three stages during this project:
- Early CML
- mCML
- Integrated Knowledge Representation

The balance of this section discusses these stages in detail.

7.1.1.1 CML Background Concepts

The original idea for implementing CML was to take work done on the Semantic Web, and on describing services within that web, and apply it to cognitive services. This work springs from DAML (DARPA Agent Markup Language) and OIL (Ontology Inference Layer), fused into OWL: Web Ontology Language. OWL is a W3C spec (http://www.w3.org/TR/owl-ref/). An instantiation of OWL for describing web services is OWL-S. OWL-S is produced by the DAML Services Coalition (http://www.daml.org/services/owl-s/1.0/). See especially the technical overview paper and the informal grounding presentation.

Another facet of CML in its original concept was probabilities. Probabilistic features in CML are a way of representing uncertainty in the knowledge representation and the space of actions. The advantages are that the application layer can concisely describe probability distributions in knowledge and, for example, express its requests as utility maximization problems.

Some recent work that develops these ideas into a language is that of Pfeffer in the language IBAL. This work grew out of earlier research in Bayesian networks and Probabilistic Relational Models [Pfeffer01].

This has been followed up by work in the programming languages research community to give it a formal basis [Ramsey02].

A language-based approach combines two features: a means for conveniently expressing probabilities and a means of specifying action (computation). OWL is good basis for encoding knowledge in a framework (e.g., categories and relationships), but it might benefit from this kind of enhancement for the purposes of:
1. Representing uncertainty in the framework, and
2. Allowing programming rather than just problem specification

Early CML included support for a probability field in the basic chunk data type. A sample interpretation of this field was as a confidence level expressed as a probability that the chunk was true. When chunks were used to perform inference, the probabilities would be combined in the expected way to yield a confidence value for the result (itself expressed as a chunk).

7.1.1.2 CML Design Outline

As CML concepts were explored, the language was pulled in two directions. First, CML was being designed for use by a cognitive application developer, so it included concrete syntax and other features appropriate to a human programmer. This led in the direction of a high-level, expressive programming language. As a practical matter, it was assumed that the application programmer would use general-purpose programming to stitch together cognitive operations, as well as possible non-cognitive operations formulated as library calls implemented directly on the hardware. Second, CML was being used as the common internal language for communication between the cognitive components. This led in the direction of a low-level, syntactically simple, declarative-style language.

The application developer mode of CML became known as "App-level CML," or just "App-CML." The Cognitive layer mode of CML became known as "Cog-level CML," or "Cog-CML." In the next sections we discuss both of these in turn.

Figure 61 shows an overview of the CML architecture.

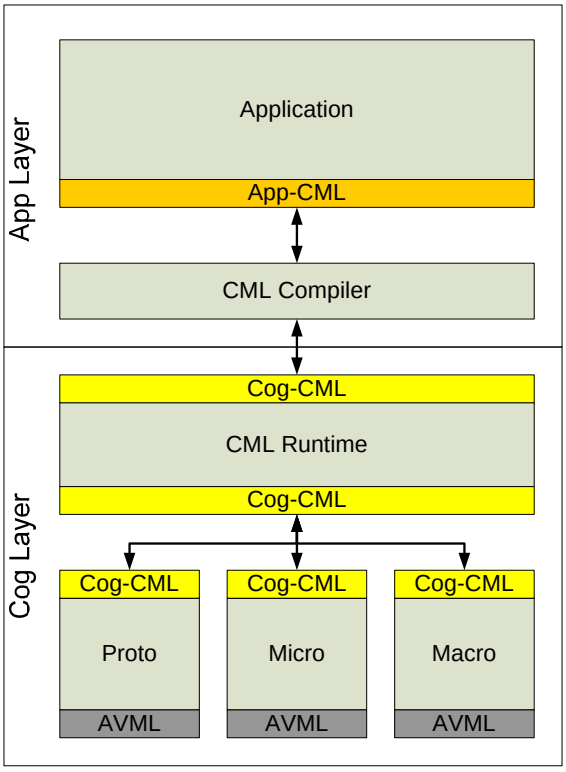


Figure 61. CML Architecture

7.1.1.3 App-level CML

CML contains features of traditional programming languages, as well as features and libraries special to the domain of cognitive applications. These special features fall into two general categories:

- Cognitive engine interface
- Modeling support

A principle requirement of App-CML is that it supports interaction with the cognitive layer. CML thus contains datatypes and special libraries for communicating with the cognitive layer. The application can define problems, submit them to the cognitive layer, and receive solutions and other results. The cognitive layer acts effectively like a service.

Modeling support allows the programmer to more concisely describe some problem or data than could be described in the general-purpose language, or to describe it in a way that can be handled by special-purpose solution engines such as the cognitive layer. In addition to providing problem-specific datatypes, CML includes a probabilistic sublanguage useful for describing (modeling) stochastic information.

7.1.1.3.1 Traditional Language Base

CML is based on the programming language ML [Milner90]. ML is a strongly-typed, high-order functional language. Features of ML include:

Infix syntax: expressions in ML are similar to those in other conventional languages. For example, y cos(2x) is written y * cos(2 * x) .

- Garbage collection: non-scalar values are created by constructors, but never need to be explicitly freed.
- User-defined datatypes: for representing the data of the problem domain, the user defines types, and then creates values of those types. This takes advantage of a well-understood language feature and has a strong correspondence with the ontological formulation used in Cog-CML. For example:

```
datatype Entity = EntityId * EnitityType * EntityName * List
EntityAttribute
type EntityId = String
type EntityType = String
...
val holland_queen = Entity("Entity1", "ship", "Holland Queen", …)
val amsterdam = Entity("Entity2", "city", "Amsterdam", …)
```

For details on ML, see the specification referenced above, or one of many good references [Lauer93] [Milner97].

7.1.1.3.2 Command Interface

The workflow for interacting with the cognitive layer is to construct a problem, submit it to the cognitive layer, and wait for a solution. Procedures for doing this are defined in the standard library, and include the following:

- BeginProblem() → problem id : int
  This procedure signals that a new congitive problem is being defined. The problem id result is a unique number that can be used to refer to the problem.

  Subsequent to this command, the data should be specified in AddChunk calls.

- EndProblemStreamingData(problem id : int)
  This procedure signals the end of the majority of chunk data for a problem. It must be called before any solutions are requested.

- RequestSolution(problem id : int)
  Requests that the Cog layer send a "best guess" solution immediately. This is not necessarily the final solution, and the Cog layer will continue to work towards a better solution.

- RequestSolutionContinually(problem id : int, interval : int)
  As for RequestIntermediateSolution, but requests periodic updates at fixed intervals. These updates will continue until this procedure is called again, with an interval of -1 meaning stop the periodic updates.

- ReceiveSolution(problem id : int) → solution : solutionType
  This procedure returns the current solution to the indicated problem.

- EndProblem(problem id : int)
  This procedure signals the end of the indicated problem. No further information about this problem will be requested, so the Cognitive layer may release any resources related to solving it, except the chunk data. Chunk data (Cog working memory) is not affected by this command.

  Part of the command interface deals with exchanging information about the raw knowledge base that forms the meat of problems and solutions, which is expressed in terms of Chunks.

- AddChunk(chunk : chunkType)
  Adds a Chunk to the knowledge base of the current problem.

- RemoveChunk(chunk id : int)
  Removes the indicated chunk from Cog layer working memory for the current problem. (The chunk value in the CML program itself is not affected.)

- UpdateChunk(chunk : chunkType)
  Changes an existing chunk. The ChunkId field of the argument must match that of a chunk that was previously added. The effect is as if that chunk was removed and the one in the argument added.

- GetChunk(chunk id : int) → chunk : chunkType
  Returns the chunk with the specified ChunkId.

7.1.1.3.3 Ontology Interface

Complementary to the commands for interacting with the cognitive layer are the datatypes that are used to encode problem data. In CML, these are realized as ML datatypes, as part of the standard library. The basic unit of information for describing data to the cognitive layer is the chunk. A special Chunk datatype is defined to encode common aspects of the information.

```
chunkType =
 Chunk of
 { ChunkId : chunkIdType,
  Attributes : attributeType list,
  Probability : real,
  ChunkContents : chunkContentsType }
```

A Chunk contains a unique identifier, and list of freeform attributes, a probability (this field is a placeholder for future expansion), and some domain-specific contents.

Problems are constructed using domain-specific ontologies. Datatypes for these ontologies are predefined in libraries, so they can be used to talk about cognitive problems and the solutions produced by the cognitive layer. It is intended to allow such libraries to be added easily, to allow additional problem domains to be addressed.

Specific ontologies are defined for the target problems of the PCAA project. An example domain of interest is intelligence analysis. Here, there are two primary kinds of data: entities (which are analogous to nouns) and events (verbs).

An entity consists of a name and a kind.

```
entityType =
{ EntityName : string,
 EntityKind : string }
```

An event consists primarily of an agent, and an action (kind). There are also fields for the target or object of the event, the event location, and its time (start and extent).

```
eventType =
{ EventAgent : chunkIdType,
 EventAction : string,
 EventObject : chunkIdType,
 EventLocation : chunkIdType,
 EventStart : eventDateType,
 EventExtent : eventExtentType }
```

7.1.1.3.4 Correspondence with Cog-CML

It is common for ontological information to be expressed in XML, and this is in fact the design decision made in Cog-CML. XML itself is simply a syntax, however, so the set of valid XML documents is typically described by a restriction language such as XML Schema. Thus the ontology is essentially the XML Schema. In CML, the role of the schema is filled by the type system, and there is a close correspondence between the two formalisms. It is interesting to examine this correspondence briefly.

In XML Schema, there are two basic kinds of types: simple (xsd:simpleType) and complex (xsd:complexType). There are additional complexities such as xsd:complexTypes that have xsd:simpleContent, but these do not affect the fundamental structure of the correspondence.

Simple types generally map well to primitive types in ML, such as int and string. Schema simple types can also be defined as restrictions on other simple types, however, through the use of xsd:restrict. For example, a xsd:simpleType may be defined that allows only xsd:string values that match a certain pattern (specified with xsd:pattern). So, while the ML types can represent all of the XML Schema data, these constraints have no clear analog in ML types.

To solve this issue, we can introduce the concept of a validity-checker, which is an ML function that checks that any particular data conforms to the schema. Since the kinds of restrictions are fixed, this function can be predefined to operate over a side data structure that exists parallel to the type structure. This side structure encodes restriction-style constraints.

Complex schema types generally map to ML algebraic datatypes. Three basic nesting structures for complex types are:
- xsd:all (conjunction): a set of items in any order
- xsd:sequence (sequence): a list of items in order
- xsd:choice (disjunction): one of a group of items

Conjunctions can be modeled in ML very naturally by using record types. Records in ML are structures of tagged values, so the field tag can be made to correspond exactly to the XML element tag. Sequences can be also be modeled by using records, but we must add side-information on the correct order for use by the validity-checker. An alternate approach would be to model sequences with ML tuples, which are ordered; we would then need to add the field tags to the side structure since ML tuples do not have them. Disjunction types can be modeled in ML by datatype constructors. The constructor names can be made to correspond to the disjunction tags.

Complex schema types may also be defined via restrictions, and as with simple schema types, these restrictions must be handled in ML by side structures. One interesting case of this is the minOccurs and maxOccurs attributes in schemas. Simple uses of these attributes can be coded directly into ML datatypes:
- minOccurs=1 maxOccurs=1: trivial case (no wrapper)
- minOccurs=0 maxOccurs=1: option datatype wrapper
- minOccurs=0 maxOccurs=unbounded: list datatype wrapper

More complicated uses such as minOccurs=400 maxOccurs=500 do not fit naturally into an ML type. Of course we can represent the data simply by using a list datatype wrapper as above, but we may allow some data that is illegal under the schema. These cases can be handled by the validity-checker.
The correspondence outlined above is sufficient to handle all of the ontology used in the Sign of Crescent example problem demo. (In fact no side structure or extra validity checking was needed; the ML type-checker was sufficient.) Other features of XML Schema can be handled

similarly. For example, we have not discussed XML attributes explicitly, but they are essentially simple types in another syntax.

We have discussed a correspondence between XML Schema and ML datatypes. This correspondence is fundamentally direct, with only a small amount of restriction information needed to represent constraints not handled by ML's type system directly. Our discussion has been informal, but we intend to make this correspondence more formal in subsequent revisions to this specification. Additionally, we intend to realize the correspondence in the form of automatic translation of XML Schema to ML datatypes.

7.1.1.3.5 Stochastic Modeling

As described in the introduction, we desire CML to have strong modeling support. CML is intended for use in describing cognitive problems concisely, and a variety of modeling features are potentially useful to that end. We choose one particular kind of modeling for inclusion in CML; while we feel this kind is especially useful, it is not necessarily the only modeling feature that could be included in the language. Our selection is both a practical instance and an exemplar of the class of modeling features.

Recently, there has been research into languages with stochastic constructs to model nondeterminism [Pfeffer01] [Ramsey02]. These languages allow concise modeling, and powerful solution techniques (e.g., variable elimination) may be applied to answer questions about the probability distributions of result values.

Rather than create an entirely new language from whole cloth, we choose to adapt an existing language. The higher-order treatment of values as distributions in much of the research suggests use of a functional language. As described previously, we have chosen ML for this base. The amount of new syntax needed is very small: we simply need a way to inject nondeterministic choice. A convenient form of this construct is `dist`, which has this form:

```
dist [ e1 : p1, ..., en : pn ]
```

The dist construct consists of a list of expression-probability pairs. Each e_i is a program expression, while each p_i is a decimal probability. The probabilities p1 through pn must all sum to 1. The expressions must be either equality types or structure types with equality types at all leaves of the nested type structure.

The semantics of CML language does not encode probabilities directly; values in CML are the same as values in ML. Execution of a CML expression (or program) simply yields one of several possible values with certain derived probabilities.

Given a probabilistic program, it is natural to ask questions not just about the "answer" of the program, but about the probability distribution of that answer. Some common evaluations are:

- Sample: A trial (e.g., sample) run of the program yields a value that is returned.
- Support: The set of possible results with nonzero probability is returned.
- Expectation: The average result is returned.

Except for "sample," most kinds of queries lead to difficult implementation issues. Answering a query may depend on a particular analysis of the program. An analysis may not be tractable for all programs, so as a result a query may not be feasibly answerable for any program. For example, in a program that is isomorphic to a Bayesian network, variable elimination can be used to solve for the variable distributions, so we call such programs solvable with respect to the expectation query. The sublanguage that allows queries of a given kind to be answered efficiently is similarly called the solvable sublanguage.

In order to solve a CML program for expectation and support, the program must be analyzable and thus cannot contain the full Turing-completeness power, so we disallow unbounded recursion (or iteration). We achieve this effect simply by allowing only first-order types, and not including any recursive constructs in the language.

7.1.1.3.5.1 Bayesian Example

To illustrate the conceptual use of the stochastic language features, we present a simple example in the domain of intelligence analysis. This example is probably not realistic, but demonstrates the stochastic features.

Suppose that we are trying to build a model of terrorist activity. Of course, it is not known a priori whether a person is a terrorist, but their actions can provide evidence of this fact. The strongest evidence is that they are found to commit a terrorist act, but the possibility also increases if they simply talk to another terrorist. The CML programmer can translate notions of the strength of such evidence into probabilities.

```
datatype object = Bomb | Terrorist | Civilian
datatype action = Explode | Communicate| Nothing
fun isTerrorist x y =
 case (x, y)
  of (Explode, ) => flip 0.9
   | (Communicate, Terrorist) => flip 0.2
   | => flip 0.1
```

(Here `flip p` is just syntactic sugar for `dist [p:true, 1-p:false]`.) Now we can apply `isTerrorist` to the action of a person, and the possibility that they are a terrorist is represented implicitly. This example could be generalized to a list of actions, with the probabilities being combined.

7.1.1.3.6 CML Standard Library

It is expected that CML will include a standard library for use by the application programmer. For example, each problem kind will have its own library, e.g., the plan recognition library, and the route planning library. These will describe the structure (types) used by the problem, and possibly also provide common facts / data. The library is itself simply written in App-CML. The contents of the library do not affect the core language definition or the CML translator.

7.1.1.4 Cog-level CML

Cog-CML was initially conceived as a message-oriented language using XML syntax. The initial version of the CML specification (version 0.1) described the principle components of the infrastructure:

- A `Message` format is used to encode all communication between cognitive components and between the App layer and the Cog layer
- A `Service` format is used to describe capabilities within the Cog layer.
- Facts are encoded as elements of the `Chunk` type.
- A `Control` Protocol defines administrative functions within CML, such as service registration and message subscription.

CML uses an XML representation, so these were defined originally as DTDs. Soon afterwards, all types were re-encoded in XML Schema.

7.1.1.5 Message Format

All communication between the Cog layer and the App layer, and between Cog components, is in the form of *messages*.

Every message has a header, which is used for administrative purposes. The intent is that the header contains transport-level information that is not relevant to the contents, and conceptually may be discarded after delivery. The rest of the message, the body (the `PCDATA token below`), forms the content. Tags that can be used as message bodies are indicated with a [M] annotation in the description.

```
<Message>
 <Header>
  <MsgSourceId>...</MsgSourceId>
  <MsgId>...</MsgId>
  <MsgTime>...</MsgTime>
  <MsgAttr>...</MsgAttr>
 </Header>
 PCDATA
</Message>
```

Message
Unique top-level tag for all messages in CML.

Header
Administrative content for a message.

MsgSourceId
Sender of this message. As indicated, this will be a ServiceId.

MsgId
Unique identifier per message per source. To get a globally unique identifier, it is necesary to prepend the MsgSourceId.

MsgTime
Time this message was sent.

MsgAttr
Miscellaneous attributes for message. This is a catch-all place where data can be stored, but it should only be used sparingly!

7.1.1.6 Services

Cog components, also called Services in this document, are declared to the system. Some services are predefined; there is also support for ad-hoc services to make themselves known. The Application layer is considered a predefined service. The Cog layer also has a predefined service for use by entities that don't have a more specialized service target. *There is a startup Protocol to be defined with the Application layer, but for now treat it as special in that it does not have to register*.

(Note: from now on we omit the Message wrapper in the outlines.)

```
<Service>
 <ServiceId>...</ServiceId>
 <ServiceName>...</ServiceName>
 <ServiceIsRequired>...</ServiceIsRequired>
</Service>

<Request>
 <RequestId>...</RequestId>
</Request>

<Response>
 <ResponseId>...</ResponseId>
</Response>
```

Service
Declares the existence of a Cog component.

ServiceId
Integer. Unique identifier for a service.

ServiceName
Human-readable name for Cog service.

ServiceIsRequired
Boolean. "true" means the Cog layer should wait until this service has registered before reporting the Cog layer ready to external parties, while “false” means do not wait. Services for which this element is true should use **RegisterServiceById**.

7.1.1.6.1 Facts

The great majority of messages exist to provide data about a request (or solution). Some of this data is background data that is not specific to any one request. Other data is specific to a kind of request.

The data that the Cog layer has to work on takes the form of chunks. At the top-level is a Chunk type; below this (in place of the `PCDATA` token used below) are domain-specific types.

```
<Chunk>
 <ChunkId>...</ChunkId>
 <ChunkRawData>...</ChunkRawData>
 <Score>...</Score>
 <Probability>...</Probability>
 PCDATA
</Chunk>
```

Chunk [M]
Message-level wrapper for a fact. General information relevant to any fact is encoded at this level. The different kinds of chunk contents form an enumeration, *which should be specified, but is currently just a PCDATA*.

ChunkId
Unique chunk identifier. (This should encode the originator to resolve the global indetifier issue.)

Score
A multipurpose score attribute. This could represent activation, or any other useful and prominant value.

Probability
Probability of this fact being true. The value is either a number between 0 and 1 or a reference to App-CML code.

7.1.1.7 Control Protocol

The control Protocol defines the order in which messages may be sent. For example, it does not make sense to ask for an answer to a problem before specifying the problem.

7.1.1.7.1 Startup Protocol

At startup, a Cog service must register with the CML engine, to indicate it is ready to accept work. When all required Cog components have registered, the Cog layer informs the application layer that the Cog layer as a whole is ready to accept work.

```
<RegisterService>
 <Service>...</Service>
</RegisterService>

<RegisterServiceById>
```

```
 <ServiceId>...</ServiceId>
</RegisterServiceById>

<CogLayerReady />
```

RegisterService [M]
Sent by a Cog service at startup to indicate it is ready to operate.

RegisterServiceById [M]
Sent by a Cog service at startup to indicate it is ready to operate. The Cog service must be one of the predefined services, which have preallocated IDs that the CML engine knows *a priori*.

CogLayerReady [M]
This message indicates to the recipient (usually the application layer in the PCAA architecture) that the Cog layer is ready to operate.

7.1.1.7.2 Publish/Subscribe Protocol

Services can set up a subscription so they only receive messages of interest. It is also possible to actively request facts from the CML database. Filtering of messages is based on XPath queries (*specifics to be determined*).

```
<Subscribe>
 <SubscribeFilter>...</SubscribeFilter>
</Subscribe>
```

Subscribe [M]
The sender wishes to receive messages that match the given pattern.

SubscribeFilter
Subscription filter in XPath form.

7.1.1.7.3 Problem Protocol

This group of messages is used to present a problem and obtain a solution. A service first sends a problem definition, potentially streaming the data for the problem as it is generated. The target works on the problem and generates a solution message when it has one, or reports intermediate results upon request.

```
<BeginRequest>
 <Request>...</Request>
 <RequestTargetId>...</RequestTargetId>
</BeginRequest>

<EndDataStreaming>
 <RequestId>...</RequestId>
</EndDataStreaming>

<RequestIntermediateResult>
 <RequestId>...</RequestId>
</RequestIntermediateResult>
```

```
<RequestIntermediateResultContinually>
 <RequestId>...</RequestId>
 <Interval>...</Interval>
</RequestIntermediateResultContinually>

<ReportIntermediateResult>
 <Response>...</Response>
</ReportIntermediateResult>

<RequestFinalResult>
 <RequestId>...</RequestId>
</RequestFinalResult>

<ReportFinalResult>
 <Response>...</Response>
</ReportFinalResult>

<EndRequest>
 <RequestId>...</RequestId>
</EndRequest>
```

BeginRequest [M]
This message is sent by a requestor to a Cog service. (Recall that this can be the Cog layer as a whole.)

RequestTargetId
The identifier of the target of the request, e.g., the service that is being asked to fulfill it.

EndDataStreaming [M]
The requestor is done streaming facts related to the request. Facts may still be sent, but they will be a result of changing requirements or situations.

RequestIntermediateResult [M]
The original *(allow others also?)* requestor can ask for an intermediate result. The target may or may not have any such result, but it should reply.

RequestIntermediateResultContinually [M]
As for RequestIntermediateResult, but the target service should keep sending intermediate results as it gets them.

Interval
How often to send intermediate results.

ReportIntermediateResult [M]
A service that is working on a request may supply an intermediate solution via this message.

RequestFinalResult [M]
The requestor signals the target service that it needs an response to this request, and this will be the last (or only) version.

EndRequest [M]
The requestor signals it is no longer interested in receiving any more messages about this request.

7.1.1.8 App-level CML

While Cog-CML was encoded in XML, for App-CML a more user-friendly language was desired. For the initial version of App-CML we chose a representation based on Standard ML [Milner90]. This had a number of advantages, including readability, but most significantly ML is a general-purpose programming language.

A design goal of App-CML was that it have a natural correspondence with Cog-CML, so that the translation of meaning would be straightforward. This was achieved by establishing a strong correspondence between ML datatypes and XML Schema datatypes. All of the XML Schema datatypes in the Cog-CML specification were encoded as ML datatypes. For example, the Chunk datatype in Cog-CML is

```
<xsd:element name="Chunk"> <!-- probably problem-independent -->
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="ChunkId"/>
   <xsd:element ref="Attribute" minOccurs="0" maxOccurs="unbounded"/>
   <xsd:element name="Probability" type="xsd:float" minOccurs="0"/>
   <xsd:element ref="ChunkContents"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
The corresponding App-CML datatype was defined as
datatype
 chunkType =
  Chunk of
   { ChunkId : chunkIdType,
    Attributes : attributeType list,
    Probability : real,
    ChunkContents : chunkContentsType }
```

A program was written to automatically translate from App-CML to Cog-CML. Due to the strong correspondence, translation was a simple recursive procedure on the structure of the data.

The early form of CML was exercised on a demo problem, Sign of the Crescent (SoC). The SoC problem data was encoded in App-CML, and then fed through the translator to Cog-CML, where it was distributed to the different components of the Cog layer (Proto-, Micro-, and Macro Cognition).

To illustrate the form of concrete data, here is an App-CML chunk from the SoC problem:

```
val entity1 =
 Chunk {ChunkId=1001,
     Attributes=[("Subtype", "container ship")],
     Probability=1.0,
     ChunkContents=
```

```
        (Entity {EntityName="Holland Queen",
               EntityKind="Ship"}) };
```

The corresponding Cog-CML chunk is:

```
<Chunk>
 <ChunkId>1001</ChunkId>
 <Attribute>
  <AttributeLabel>Subtype</AttributeLabel>
  <AttributeValue>container ship</AttributeValue>
 </Attribute>
 <Probability>1.0</Probability>
 <ChunkContents>
  <Entity>
   <EntityName>Holland Queen</EntityName>
   <EntityKind>Ship</EntityKind>
  </Entity>
 </ChunkContents>
</Chunk>
```

7.1.2 mCML

The next major step in the evolution of CML was the adoption of a different formalism for programming at the Application layer. This declarative form (which is not covered in this document) was given name CML but is otherwise unrelated to the formalisms described in this section. The earlier form of CML, specifically the version for Cog-level CML, was retained for use within the Cog layer itself and as an interface, and renamed mCML.

mCML has two roles (Figure 62):

- mCML is the internal language of the Cog layer. The Cog components use it as a common knowledge representation and as a common language for inter-component communication.
- mCML is the interface between the App layer (new CML) and the Cog layer. mCML represents problem knowledge (data + structure) and solutions. mCML also represents the service/task structure of the Cog layer for use by the App layer.

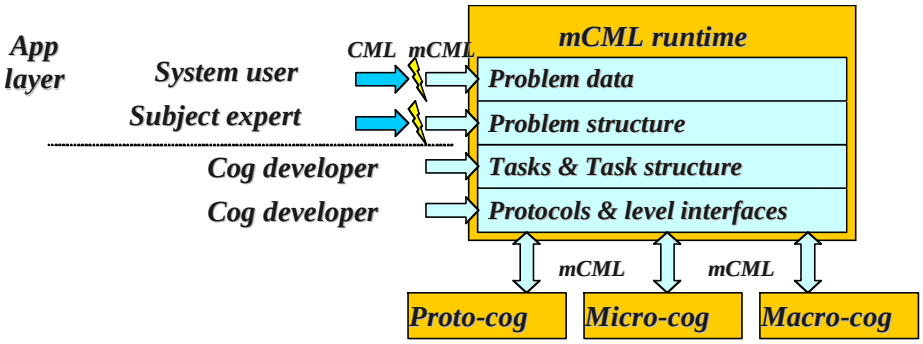


Figure 62. mCML Architecture

Internally, mCML is not just a language for describing domain data, but also problem structure and the Protocol for interaction between the Cog components. Thus mCML message types can generally be organized into three categories:

1. **Knowledge management.** mCML messages such as `AddChunk` and `RemoveChunk` allow management of the problem data and general domain knowledge.
2. **Problem and task structure.** mCML messages such as `DeclareGT` describe the structure of problems, and are also used to encode requests between Cognitive components.
3. **Operational Protocols.** From an administrative perspective, Protocols are needed to specify how to give problems to the Cog layer (`BeginProblem`) and how to deliver Solutions back to the Application layer (`ReportSolution`), as well as other basic operations.

The set of messages in mCML is fixed and general across problem domains. However, the structure of data contained in them, and the protocols used to access them is problem-specific. For example, the `AddChunk` message is always used to data, but the structure of that data within the message is often problem-specific. Thus mCML is designed to be modular; there is a generic part into which domain-specific parts can be plugged (Figure 63).

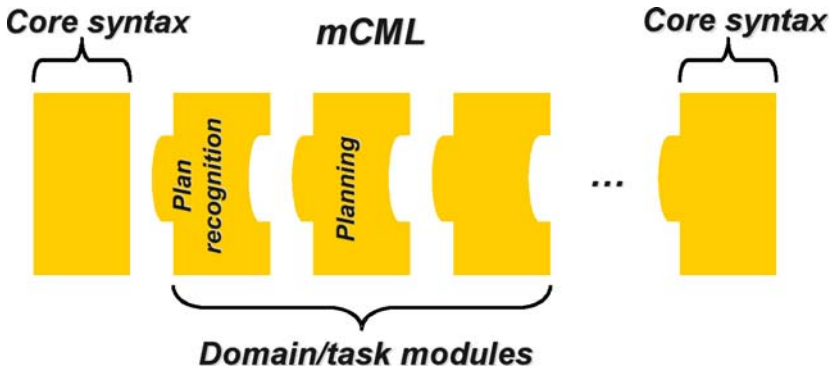


Figure 63. Pluggable mCML

In addition, there is an active aspect of mCML, called the mCML Runtime. The mCML Runtime acts as a common repository for Cog layer data, and also manages message traffic between Cog components.

7.1.2.1 mCML Specification

The normative definition of mCML is an XML Schema is given below. Though a number of details have changed, the structure is essentially similar to that in earlier versions of Cog-CML.

The key message types are as follows:

- **Knowledge Management**
  - AddChunk: Add a chunk to the Cog layer knowledge store.
  - RemoveChunk: Remove a chunk from the Cog layer knowledge store.
  - UpdateChunk: Change the contents of a chunk.
  - GetChunk: Request a chunk from the store.
  - OldChunk: Reply to a request.
- **Problem and task structure**
  - DeclareGT: Declare a generic task.
  - MicroRequestInference: Request an inference from Macro.

  - MacroNotifyInference: Reply with an inference to Micro.
  - ProtoNotifySimilarity: Notify Micro and Macro about a similarity cluster.
  - MicroNotifyGoal: Determine a goal.
- **Operational**
  - BeginContext: Application layer message opening a context in the Cog store for holding related data.
  - EndContext: Close the context; the data is no longer relevant.
  - StartGT: Application layer message requesting Cog layer to begin working on a task.
  - UpdateGT: Allows parameters of currently-active GT to be modified.
  - EndGT: Application layer message indicating it is finished with the indicated GT.
  - EndStreamingData: Advisory message from application layer indicating the context now has most of the data that will be streamed into it.
  - RequestSolution: Application layer message requesting a solution, indicating the timeframe and whether it will be an intermediate request to be refined later or the final answer.
  - RequestSolutionContinually: Application layer message requesting solutions be delivered and updated automatically according to the specified schedule.
  - ReportSolution: Cog layer message delivering a solution to a GT.

An outline of the typical communication pattern between the Application layer and the Cog layer is as follows:

```
 <BeginContext … />           <!-- Cog must prepare for batch of data -->
 <AddChunk … /> …       <!-- problem data specified in AddChunk's -->
 <EndContextStreamingData/> <!-- Main data transfer complete -->
 <StartGT … />                <!-- App indicates which task to work on -->
 <AddChunk … /> …       <!-- dynamic task adjustment via new data -->
 <RequestFinalSolution … />   <!-- App wants a single answer, and when -->
 <ReportFinalSolution … />    <!-- Cog provides an answer -->
 <EndGT/>                     <!-- App is done with this task -->
 <EndContext/>                <!-- Cog may release data; no new GTs on it -->
```

7.1.2.1.1 mCML Specification

The following is the normative definition of mCML, expressed as an XML Schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- ================================================================
   mCML.xsd : XML Schema for core Cog CML (mCML)
 -->

 <xsd:annotation>
  <xsd:documentation xml:lang="en">
   mCML schema
  </xsd:documentation>
 </xsd:annotation>

<!-- ================================================================
   Schema for problem-specific constructs
 -->
```

```
 <xsd:include schemaLocation="mCML_GT.xsd"/>
 <xsd:include schemaLocation="mCML_PR.xsd"/>
 <xsd:include schemaLocation="mCML_RP.xsd"/>
 <xsd:include schemaLocation="mCML_AVM.xsd"/>

<!-- ================================================================
   Basic data and element types
 -->

 <!-- integer type with an explicit type attribute -->
 <xsd:complexType name="AttInt">
  <xsd:simpleContent>
   <xsd:extension base="xsd:integer">
    <!-- <xsd:attribute name="type" type="xsd:string" use="required"
fixed="integer"/> -->
    <xsd:attribute name="type" type="xsd:string" fixed="integer"/>
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>

 <!-- unsigned integer type with an explicit type attribute -->
 <xsd:complexType name="AttUInt">
  <xsd:simpleContent>
   <xsd:extension base="xsd:unsignedInt">
    <!-- <xsd:attribute name="type" type="xsd:string" use="required"
fixed="integer"/> -->
      <xsd:attribute name="type" type="xsd:string" fixed="integer"/>
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>

 <!-- float type with an explicit type attribute -->
 <xsd:complexType name="AttFloat">
  <xsd:simpleContent>
   <xsd:extension base="xsd:float">
    <!-- <xsd:attribute name="type" type="xsd:string" use="required"
fixed="double"/> -->
    <xsd:attribute name="type" type="xsd:string" fixed="double"/>
   </xsd:extension>
  </xsd:simpleContent>
 </xsd:complexType>

 <xsd:complexType name="EmptyType">
 </xsd:complexType>

 <xsd:element name="AttributeLabel" type="xsd:string"/>
 <xsd:element name="AttributeValue" type="xsd:anyType"/>
 <xsd:element name="Attribute" type="AttributeType"/>
 <xsd:complexType name="AttributeType">
  <xsd:sequence>
   <xsd:element ref="AttributeLabel"/>
   <xsd:element ref="AttributeValue"/>
  </xsd:sequence>
 </xsd:complexType>
```

```
<!-- ====================================================================
   Message format
 -->

 <xsd:element name="Message">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="Header"/>
    <xsd:choice>

     <xsd:element ref="AddChunk"/>
     <xsd:element ref="RemoveChunk"/>
     <xsd:element ref="UpdateChunk"/>
     <xsd:element ref="GetChunk"/>
     <xsd:element ref="OldChunk"/>

     <!-- ==== GT / service ==== -->
     <!-- Old-style, hardcoded GTs -->
     <xsd:element ref="ProtoNotifyCompatibility"/>
     <xsd:element ref="ProtoNotifySimilarity"/>
     <xsd:element ref="ProtoNotifyBestPaths"/>
     <xsd:element ref="ProtoNotifyLocationClusters"/>

     <xsd:element ref="MicroNotifyGoal"/>
     <xsd:element ref="MicroNotifyCompatibility"/>
     <xsd:element ref="MicroRequestInference"/>
     <xsd:element ref="MicroRequestRoute"/>

     <xsd:element ref="MacroNotifyInference"/>
     <xsd:element ref="MacroNotifyGoal"/>
     <xsd:element ref="MacroNotifyRoute"/>

     <!-- New-style GT declaration -->
     <xsd:element ref="DeclareGT"/>

     <!-- Cog level registration -->
     <xsd:element ref="RegisterService"/>
     <xsd:element ref="RegisterServiceById"/>
     <xsd:element ref="CogLayerReady"/>

     <xsd:element ref="BeginContext"/>
     <xsd:element ref="EndContext"/>
     <xsd:element ref="StartGT"/>
     <xsd:element ref="UpdateGT"/>
     <xsd:element ref="EndGT"/>
     <xsd:element ref="EndStreamingData"/>
     <xsd:element ref="RequestSolution"/>
     <xsd:element ref="RequestSolutionContinually"/>
     <xsd:element ref="ReportSolution"/>
     <xsd:element ref="ReportSolutionApp"/>

     <!-- diagnostic/measurement messages -->
     <xsd:element ref="ResourceUsage"/>
     <xsd:element ref="ResetResourceUsage"/>
```

```
      <!-- AVM -->
      <xsd:element ref="ACIPLMatchData"/>
      <xsd:element ref="ACIPLMatch"/>

      <!-- temporary(?) messages -->
      <xsd:element ref="ClearAll"/>
      <xsd:element ref="ResetAll"/>
      <xsd:element ref="Shutdown"/>

    </xsd:choice>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="Header">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="MsgId" type="xsd:unsignedInt"/>
    <xsd:element name="MsgSourceId" type="xsd:string"/>
    <xsd:element name="MsgDestinationId" type="xsd:string" minOccurs="0"/>
    <!-- <xsd:element name="MsgTime" type="xsd:dateTime"/> -->
    <xsd:element name="MsgTime" type="xsd:string"/>
    <xsd:element name="Comment" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- for managing groups of messages -->
 <xsd:element name="MessageList">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="Message" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

<!-- ================================================================
   Facts
 -->

 <xsd:simpleType name="ChunkIdType">
  <xsd:restriction base="xsd:unsignedInt">
  </xsd:restriction>
 </xsd:simpleType>
 <xsd:element name="ChunkId" type="ChunkIdType"/>

<!-- ================================================================
   Static entities
 -->

 <xsd:element name="ServiceId" type="xsd:unsignedInt"/>

 <xsd:element name="Service">
  <xsd:complexType>
   <xsd:sequence>
```

```
    <xsd:element ref="ServiceId"/>
    <xsd:element name="ServiceName" type="xsd:string"/>
    <xsd:element name="ServiceIsRequired" type="xsd:boolean"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="Solution">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="ResultPair" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="Tag" type="xsd:string"/>
       <xsd:element name="Value" type="ChunkIdType"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="GTId" type="xsd:unsignedInt"/>

 <xsd:element name="ContextId" type="xsd:unsignedInt"/>

 <xsd:element name="Context">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Id" type="xsd:unsignedInt"/>
    <xsd:element name="Parent" type="xsd:unsignedInt" minOccurs="0"/>
    <xsd:element name="Name" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

<!-- ================================================================
   Control messages (startup)
 -->

 <xsd:element name="RegisterService">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="Service"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="RegisterServiceById">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ServiceId"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
```

```
<xsd:element name="CogLayerReady">
  <xsd:complexType>
  </xsd:complexType>
 </xsd:element>

<!-- ====================================================================
   Chunk messages
 -->

 <xsd:element name="AddChunk">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="Chunk" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="RemoveChunk">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="UpdateChunk">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="Chunk" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Requests sending of an existing chunk -->
 <xsd:element name="GetChunk">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Reply to a GetChunk request -->
 <xsd:element name="OldChunk">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="Chunk" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

<!-- ====================================================================
   Proto-Cog API
 -->
```

```
 <!-- Notify Micro cognition that a group of compatibility chunks
    has just been posted. References are included in this msg. -->
 <xsd:element name="ProtoNotifyCompatibility">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
<!-- Notify Micro cog that a group of similarity chunks has just
    been posted. References are included in this msg. -->
 <xsd:element name="ProtoNotifySimilarity">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Notify Micro about the best paths from one location.
    References to RoutePlan chunks are included in this msg. -->
 <xsd:element name="ProtoNotifyBestPaths">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Notify Micro about clusters of Targets (& other elements).
    References to Target/Threat/Waypoint chunks are included in
    this msg. -->
 <xsd:element name="ProtoNotifyLocationClusters">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

<!-- ================================================================
   Micro-Cog API
 -->

 <!-- Send current goal chunk from MIC goal buffer (to MAC) -->
 <xsd:element name="MicroNotifyGoal">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Send new compatibility chunk from MIC memory buffer (to PRO) -->
 <xsd:element name="MicroNotifyCompatibility">
```

```
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="MicroRequestInference">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Premise" type="ChunkIdType" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Do we need another control msg here to publish "speculative
    trifles"? Or can we just use the regular AddChunk message? -->

 <xsd:element name="MicroRequestRoute">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Start" type="ChunkIdType"/>
    <xsd:element name="End" type="ChunkIdType"/>
    <xsd:element name="LocationCluster" type="ChunkIdType"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

<!-- ================================================================
   Macro-Cog API
 -->

 <!-- Notify Micro cog that an inference has just been posted. -->
 <xsd:element name="MacroNotifyInference">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Premise" type="ChunkIdType" maxOccurs="unbounded"/>
    <xsd:element name="Assertion" type="ChunkIdType" maxOccurs="unbounded"/>
      <xsd:element name="Description" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Maybe not needed, but suggested as possibly useful by Christian -->
 <xsd:element name="MacroNotifyGoal">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="MacroNotifyRoute">
  <xsd:complexType>
   <xsd:sequence>
```

```
    <xsd:element name="Start" type="ChunkIdType"/>
    <xsd:element name="End" type="ChunkIdType"/>
    <xsd:element name="LocationCluster" type="ChunkIdType"/>
    <xsd:element name="RoutePlan" type="ChunkIdType"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

<!-- ================================================================
   Control messages to/from App layer
 -->

 <xsd:element name="BeginContext">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="Context"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- Allows deletion of all the Chunks contained in this context. -->
 <xsd:element name="EndContext">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ContextId"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="StartGT">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="GTId"/>
    <xsd:element ref="ContextId"/>
    <xsd:element name="Deadline" type="xsd:dateTime" minOccurs="0"/>
    <xsd:element name="GTCall" type="GTCallType"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- allows any or all parameters to be updated -->
 <xsd:element name="UpdateGT">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="GTId"/>
    <xsd:element name="Deadline" type="xsd:dateTime" minOccurs="0"/>
    <xsd:element name="GTCall" type="GTCallType"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="EndGT">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="GTId"/>
```

```
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:element name="EndStreamingData">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="ContextId"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:element name="RequestSolution">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="GTId"/>
   <xsd:element name="Deadline" type="xsd:dateTime" minOccurs="0"/>
   <xsd:element name="Final" type="EmptyType" minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:element name="RequestSolutionContinually">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="GTId"/>
   <xsd:element name="Interval" type="xsd:unsignedInt"/> <!-- seconds -->
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:element name="ReportSolution">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="GTId"/>
   <xsd:element ref="Solution" minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<!-- For use by CMLRuntime to report to the App. It's the same as
   ReportSolution, but with a different name to avoid confusion.
   This allows CMLRuntime to adapt the format of the solution to
   something more easily handled by the App. App should be
   expecting this message (not ReportSolution).
 -->
<xsd:element name="ReportSolutionApp">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element ref="GTId"/>
   <xsd:element ref="Solution" minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

```
<!-- ====================================================================
   Diagnostic and measurement message types
 -->

 <xsd:element name="ResourceUsage">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Source" type="xsd:string"/>
    <xsd:element name="CumulativeOps" type="xsd:float"/>
    <xsd:element name="CumulativeCPU" type="xsd:float" minOccurs="0"/>
    <xsd:element name="PercentCPU" type="xsd:float" minOccurs="0"/>
      <xsd:element name="QualityRatio" type="xsd:float" minOccurs="0"/>
      <xsd:element name="MatchesPerSecond" type="xsd:float" minOccurs="0"/>
      <xsd:element name="IndexHit" type="xsd:integer" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="ResetResourceUsage">
  <xsd:complexType>
  </xsd:complexType>
 </xsd:element>

<!-- ====================================================================
   Temporary(?) message types
 -->

 <xsd:element name="ClearAll">
  <xsd:complexType>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="ResetAll">
  <xsd:complexType>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="Shutdown">
  <xsd:complexType>
  </xsd:complexType>
 </xsd:element>

<!-- ====================================================================
   Chunk top-level Chunk structure
 -->

 <xsd:element name="ChunkContents">
  <xsd:complexType>
   <xsd:choice>

    <xsd:element ref="Entity"/>
    <xsd:element ref="Event"/>
    <xsd:element ref="List"/>

      <xsd:element ref="Inference"/>
```

```
      <xsd:element ref="Goal"/>
      <xsd:element ref="CompatibilityCluster"/>
      <xsd:element ref="SimilarityCluster"/>

      <!-- <xsd:element ref="PDDL"/> -->
      <!-- <xsd:element ref="POP"/> -->

      <xsd:element ref="Area"/>
      <xsd:element ref="Location"/>
      <xsd:element ref="UAV"/>
      <xsd:element ref="UAVKind"/>
      <xsd:element ref="Weapon"/>
      <xsd:element ref="WeaponKind"/>
      <xsd:element ref="Target"/>
      <xsd:element ref="LocationCluster"/>
      <xsd:element ref="Threat"/>
      <xsd:element ref="ThreatKind"/>
      <xsd:element ref="Waypoint"/>
      <xsd:element ref="RoutePlan"/>
   </xsd:choice>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="Chunk"> <!-- probably problem-independent -->
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="ChunkId"/>
    <xsd:element ref="ContextId" minOccurs="0"/>
    <xsd:element ref="Attribute" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Probability" type="xsd:float" minOccurs="0"/> <!-- to
be elaborated -->
    <xsd:element ref="ChunkContents"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

</xsd:schema>
<!-- ================================================================
   End of mCML.xsd
 -->
```

7.1.3 Agent Virtual Machine Language (AVML)

The goal of AVML is to provide efficient, portable, and accessible utilization of the PCAA hardware.

- **Efficient:** Efficient utilization of the PCAA hardware means taking advantage of the particular features of the hardware. Many hardware features can be automatically used by the compiler, but one area in which the compiler often needs help is expressing the parallelism of the application.
- **Portable:** AVML should not be so dependent on the details of the architecture that it must be changed significantly when the underlying architecture changes.

- **Accessible:** AVML must remain a high-level language that is convenient for human developers. Low-level languages may offer efficient and possibly even portable access to the hardware, but if it is too low level then the size of programs that may be practically written will be sharply limited.

Cognitive applications are characterized by irregular parallelism. Cognitive applications often have fine-grained parallelism, but it is not regular, as in scientific computing when iterating over arrays. The parallelism comes from things like parallel access to data structures, often involving pointer chasing, and from fine-grained task-level parallelism.

7.1.4 AVML 1.0

We designed an AVML (which we refer to as 1.0 when necessary to avoid ambiguity) that provides a common structure for expressing cognitive components that enables experimentation with high-level strategies for data distribution, load balancing and synchronization. In essence, this AVML is based on structured threads and locks. However, AVML focuses more on describing the structure of the computation, giving a compiler and runtime system flexibility to explore and experiment with different implementation policies for coordinating parallel computation.

7.1.4.1 AVML Model

The main abstractions in the AVM model are nodes and actions. The AVM model divides the data operated on by an application into discrete units called nodes and divides the operations performed by an application into discrete units called actions.

A node is simply a name for a collection of data that an application intends to operate on as a unit. When an application allocates new data, it is assigned to an existing node or allocated to a new node. A node may reference other nodes, creating an explicit data topology. A sophisticated runtime can take advantage of the data topology to schedule computation and organize data distribution.

An action is a discrete unit of computation. It embodies a function and its arguments. An action executes atomically with respect to other actions. Each action, at creation, specifies the set of nodes that it will access. Before executing, an action gains exclusive access to the nodes it will modify in a predefined, static order. Any competing action trying to modify the same nodes must acquire the nodes in the same order, thus preventing deadlock. During the conclusion of the execution of an action, an action may create new actions and/or new nodes. A runtime is responsible for scheduling the next action to execute.

Although nodes may reference other nodes through pointers, an action is limited in the forms of dereferencing that it may perform during its execution. An action may only dereference a pointer if the pointer refers to data in a node that was specified at the creation of the action. This constraint bounds an action's data accesses during a single execution and can enable different scheduling and data distribution strategies.

The AVM model can handle general algorithms, but it is especially tailored to algorithms centered around persistent graphs, such as swarming and Rete.

7.1.4.2 AVML Implementation

We implemented a prototype version of the AVML model on top of C. Conceptually, AVML will be compiled by RStream compiler [RStream] to a low-level language consisting of C and hardware-level operations rendered as library calls. For our prototype, we chose MPI as a convenient abstraction for the hardware-level operations. AVML was implemented as a C runtime library and a set of preprocessor macros that transform AVML-C into C plus MPI calls.

7.1.4.2.1 Nodes

In AVML, a node is a collection of C structs. Nodes are created implicitly with the allocation of new memory by AVM NEW. By passing a reference to another data value, AVM NEW also allows a programmer to specify that the new memory be allocated in the node containing that value.

```
/* in a new node, allocate one value of type Item and return a pointer
to the newly allocated memory */
GPTR( Item ) item1 = AVMNEW(AVMNULL, Item , 1 ) ;

/* allocate another value in the node referenced by item1 */
GPTR( Item ) item2 = AVMNEW( item1 , Item , 1 ) ;

/* allocate an array of ten values in the node referenced by item1 */
GPTR( Item ) i tems = AVMNEW( item1 , Item , 1 0 ) ;
```

Example of using AVMNEW

Any value stored within a node must be defined using the AVML macros: AVM_BEGIN_STRUCT, AVM_END_STRUCT, DFN_GPTR, DFN_LPTR and DFN_VALUE. These macros define an AVM_STRUCT. They generate code that allows the runtime system to determine the extents of data types and to serialize them if needed. DFN_GPTR defines a global pointer which may point to any value, DFN_LPTR defines a local pointer which typically points to a value within the same node, and DFN_VALUE defines an element in the struct which is interpreted solely as a data value. An AVM_STRUCT definition should only contain the aforementioned macros.

The DFN_LPTR(T, name) macro defines a struct element of type T *, i.e., a standard C pointer. DFN_GPTR(T, name) defines an element of type GPTR(T), and DFN_VALUE(T, name) defines an element of type T.

```
AVM_BEGIN_STRUCT(Item)    /* name of the struct */
    DFN_LPTR(char, name); /* local pointer, local data */
    DFN_VALUE(int, data); /* local value */
    DFN_GPTR(Item, next); /* recursive data structures can be defined
too */
AVM_END_STRUCT()
```

Example of declaring and defining a user struct

In order to create the AVM model of data access, AVML organizes references within a node (i.e., pointers stored inside an AVM_STRUCT) into two different types, local pointers and global pointers. Local pointers are traditional C pointers. They are always valid (i.e., able to be dereferenced) when they point to another value within the same node. A local pointer to a value in another node is only valid during the particular execution of the action that created it. The validity of local pointers to values in other nodes does not persist beyond the execution of a single action. Accessing an invalid local pointer results in undefined behavior.

Global pointers, on the other hand, do maintain their validity between executions of actions even when they point to values in another node, but they can only be dereferenced when the node that contains the referenced value is owned by the dereferencing action. An action owns a node by having a reference to a value in the node as an argument to the action. Since action arguments are specified at an action's creation and before the execution of the action, the nodes accessed by an action are fixed before execution.

All AVML data should be allocated with AVM_NEW. AVM_NEW takes three arguments: a global pointer, a type and a count, and returns a global pointer to a block of memory large enough to store count values of the given type. The memory will be allocated in the same node as the value pointed to by the given global pointer.

```
PTR(Item) gp = AVM_NEW(AVM_NULL, Item , 1 ) ;

/* the next line is not valid because a global pointer cannot be
directly assigned to a local one (this property is checked statically
by AVML) */
Item * lp bad = gp ; /* Error ! */

/* however , we can explicitly cast a global pointer to a local pointer
*/
Item * lp = G2L(gp) ;

/* if the local pointer points to data with in the same node , pointer
will be valid after the end of the action */
lp->name = G2L(AVM_NEW(gp, char, 50);

/* in the following , dereferecing the value next will not be valid
after the end of the action */
GPTR(Item) gp2 = AVM_NEW(AVM_NULL, Item, 1);
lp->name = G2L(AVM_NEW(gp2, char, 50);

/* ... although the local pointer is valid until the end of the
execution of this action */
strcpy(lp->name, "some data");

/* a global pointer is valid all the time */
G2L(gp)->next = gp2;

/* ... but it can only be dereferenced when the referenced node is
owned by the currently executing action */
assert(strcmp(G2L(G2L(gp)->next)->name, "some data") == 0);
```

Example of using local and global pointers

7.1.4.2.2 Actions

To create an action in AVML, a user specifies a function to be executed and an argument that will be passed by value to the function as a parameter. The argument is an AVM_STRUCT that typically contains global pointers to nodes. It cannot contain local pointers.

At the creation of an action, additional arguments provide hints to the runtime about what order actions should be executed in, how much work the action will take to execute, and how much total work the actions descendent actions will take to execute. These values are only hints and do not guarantee a particular execution order. These values, combined with profiling, are used by the runtime for load balancing.

To execute an action, the runtime gains exclusive access to the nodes referenced in the arguments of the action to prevent any other action from modifying a value while the action is executing. There are several common methods to achieve the appearance of exclusive access: mutexes on the nodes, or eager execution followed by a rollback in cases of inconsistency. An implementation of AVM is free to use any mechanism as long as each execution of an action is given an atomic view of accessed values.

```
AVM_BEGIN_ACTION(process_item, Item, item)
    /* creating argument to action */
    GPTR(Item) arg = AVM_NEW(AVM_NULL, Item, 1);
    G2L(arg)->data = G2L(item)->data - 1;
    G2L(item)->next = arg;
    if (G2L(arg)->data > 0) {
        AVM_ACTION(process_item, arg, AVM_HINT_UNKNOWN,
AVM_HINT_UNKNOWN, AVM_HINT_UNKNOWN);
    }
AVM_END_ACTION()
```

Example of defining and creating an action

7.1.4.2.3 Garbage Collection

An AVML program must allocate all its AVML data structures using AVM_NEW. Additionally, all AVML data structures are composed of AVM_STRUCTs which, through AVML macros, have elements with known data lengths and types. Therefore, it is possible to perform automatic garbage collection during the execution of AVM_RUN by constructing the root set of data from all the argument pointers from all currently executing actions and all actions scheduled to execute. AVML implements automatic garbage collection using a periodic mark and sweep of memory.

7.1.4.3 Example: Swarming

Here we give an integrated example of AVML, solving the all-points shortest path problem. In this implementation each node in the graph is realized as a computational element (i.e., a swarm agent). Nodes communicate with their neighbors, propagating information about path distances. Below is the function that does the work at a node.

```
/* propagate pheromones from hex */
#ifdef __AVM
AVM_BEGIN_ACTION(Hex_DoWork, Work, w)
#else
void Hex_DoWork(Work* w, int* numWork, Work* workList) {
#endif
  /* don't need to do anything if hex already has at least
   * as high intensity pheromones as being propagated */
  if (G2L(w->hex)->bestValue[w->target] < w->value) {
    Work newW;
    int sa;
    int s;

    /* update hex */
    G2L(w->hex)->bestValue[w->target] = w->value;
    G2L(w->hex)->bestFromSide[w->target] = w->fromSide;

    /* create new work items to update neighbors... */

    /* attenuate pheromones */
    newW.target = w->target;
    if (G2L(w->hex)->threatCount == 0) {
      newW.value = w->value * SAFE_ATTEN;
    } else {
      newW.value = w->value * THREAT_ATTEN;
    }

    /* for target hex, propagate in all directions */
    if (w->fromSide == -1) {
      for (s = 0; s < HEX_SIDES; s++) {
        newW.hex = G2L(w->hex)->neighbors[s];
        if (!GPTR_IS_NULL(newW.hex)) {
          newW.fromSide = OPP_SIDE(s);
#ifdef __AVM
          {
            AVM_ACTION(Hex_DoWork, Work, newW,
              (int)(newW.value * 10000), AVM_HINT_UNKNOWN,
              AVM_HINT_UNKNOWN);
          }
#else
          workList[*numWork] = newW;
          (*numWork) += 1;
#endif
        }
      }

    /* for other hexes only need to propagate through side of hex
opposite
     * side pheromones arrived through */
    } else {
      for (sa = HEX_OPP_HALF_START; sa < HEX_OPP_HALF_END; sa++) {
        s = ADD_SIDE(w->fromSide, sa);
        newW.hex = G2L(w->hex)->neighbors[s];
        if (!GPTR_IS_NULL(newW.hex)) {
          newW.fromSide = OPP_SIDE(s);
```

```
#ifdef __AVM
          {
            AVM_ACTION(Hex_DoWork, Work, newW,
               (int)(newW.value * 10000), AVM_HINT_UNKNOWN,
               AVM_HINT_UNKNOWN);
          }
#else
          workList[*numWork] = newW;
          (*numWork) += 1;
#endif
        }
      }
    }
  }
#ifdef __AVM
AVM_END_ACTION()
#else
}
#endif
```

7.1.5 Related Work

AVML is a parallel programming model based on the task decomposition paradigm. It was initially influenced by the ALPS search framework [Xu05], a C++ framework for parallel search. In AVML, Actions are tasks, and a runtime system is responsible for intelligent scheduling and mapping of tasks to physical resources. The underlying machine model has been deliberately left unspecified, and AVML can be implemented on a shared memory or distributed memory machine. Certain features of AVML (e.g., nodes and the restricted dereferencing of AVML) allow efficient mapping to either architecture. AVML is designed for efficient implementation on many hardware architectures.

There have been several programming models that have bridged the gap between shared and distributed memory. Distributed shared memory (DSM) systems model global shared memory using distributed local memory. In order to facilitate better mapping to distributed memory, some DSM systems like UPC [Coarfa05] differentiate between local data and shared global data. Although both types of memory may be accessed through the uniform interface of pointer dereferencing, global data typically has a longer access time. AVML strengthens the distinction between local and global data by restricting when global pointers can dereferenced and limiting the validity of local pointers.

AVML uses task parallelism, which is based on the decomposition of computation tasks in to parallel subtasks. It is related to the structured fork/join parallelism introduced in Cilk [Frigo98] (fork/join is called spawn/sync by the authors). However, in contrast to Cilk, AVML also introduces explicit data partitioning through the node abstraction. The partitioning of both data and computation is similar to the encapsulation property held by parallel object-oriented programming systems like Charm++ [Kale93] in which computation is strongly associated with the data it operates on. AVML combines the task parallelism of Cilk with the managed data layout of UPC.

AVML can support two different methods of passing data between actions. The first is the traditional message-passing style [MPI] where one action explicitly sends data to another process through a separate address space (i.e., the address space of the message). In AVML, this occurs when one action passes data to another action through the latter action's arguments. AVML can also support a shared-memory approach where one action communicates to another action through a common address space by dereferencing global pointers and modifying shared data. Under this system, scheduling another action is akin to explicitly notifying another process that shared data has changed.

7.2 Application Programming Interfaces (APIs) and Libraries (Libs)

7.2.1 (ACIPL) Advanced Cognitive Information Processing Library

The successful development and adoption of optimized libraries like Vector Signal Image Processing Library (VSIPL) and Linear Algebra Package (LAPACK) has been crucial to progress in complex processing systems. It enables the hardware designers to create hardware directed at supporting a detailed specified functionality, and to deploy experts to hand-optimize the required functionality on the target devices. It also enables software requirement specifications and development to be implemented at higher levels of standardized representation. We take inspiration from the VSIPL approach to define a library of modules that provide algorithms and abstractions useful to cognitive computing.

Although ACIPL is designed as a collection of APIs, these API should share some common features, such as allowing anytime access. The APIs should also have a regularity of interface that makes it easy, after learning one particular API, to understand and use others.

Since the Cognitive layer is composed of three specific engines implementing the three parts of C3I1, it makes sense to include special support for the performance-critical sections of those engines in ACIPL. In addition, there will be a need for general support for cognitive-relevant algorithms. Accordingly, we organize ACIPL into Cog engine support and generalized modules.

7.2.1.1 Cognitive Engine Support

7.2.1.1.1 Proto: Swarming

We developed a conceptual ACIPL interface to Swarming. This is a simple form in which there is a "sea" of equivalent agents connected to each other in some topology. It doesn't involve hierarchy, for example. The sample API is given in Figure 64.

The major datatypes are:

- An **`ACIPL_TASK`** references a particular task in the AVM. In this case, it is a swarming computation. The computation can be created, stopped, modified, etc. by through variables of this type.
- A **`SwarmAgent`** is a computational unit in the swarm. They conceptually execute in parallel.

- **`SwarmBehavior`** is run as the activity of a swarm agent. This can be implemented as a function pointer type, where the agent invokes the function to do its work. [Details on this function pointer type are not developed yet.]
- Each agent has some state of type **`SwarmData`**. It can be manipulated directly (as in setting initial conditions or reading out results) or by the SwarmBehavior.
- Agents communicate via messages of the form **`SwarmMsg`**. Messages pass from an agent to its neighbors.

To use this API in an actual swarming system, the developer would write something that creates a swarm, populates it with agents, sets behavior for all agents, and then runs the swarm task. The AVM handles the mapping of agents to the hardware and does dynamic load balancing as the system runs. Results from a swarm are either sent out implicitly as a part of Agent behavior, or may be read out from the data of agents while the swarm is paused.

7.2.1.1.2 Micro: ACT-R

The outline of the ACT-R process is given in the previous section on hardware mapping (see Figure 65). An ACIPL abstraction for this component was not developed.

7.2.1.1.3 Macro: Soar

We developed a simple abstraction of the Soar engine in API form. This abstraction attempts to capture the essence of Soar process, but leaves out various details and issues (such as chunking and subgoaling).

Soar's basic loop has the following steps:

1. Process input
2. Match, fire, and retract productions until there is nothing left to fire
3. Make a decision (operator or impasses) by evaluating conflict set
4. Apply operators (special case of match fire)
5. Send output commands to external systems

Of these steps, by far the most expensive in current (serial) Soar implementations is processing input. This is both a process communication/bandwidth issue (mostly) and a Rete issue (less so, see below).

Soar has 3 (main) memories:

1. Production memory, a collection of productions with conditions and actions. In this sample API, production memory is static, but in actual Soar implementations that do chunking, production memory can and does change (incrementally and monotonically).
2. Short-term memory (STM), a rooted graph of id-attr-value objects. The objects are called "working memory elements" (WMEs). Short-term memory includes named buffers designated for input and ouput.
3. Preference memory, annotations of objects proposed for short-term memory. Conceptually, preference memory and STM are distinct but in actual implementation, preference memory is implemented via pointers/elaboration of the objects in STM.

```
typedef ACIPL_TASK;
typedef SwarmAgent;
typedef SwarmBehavior;
typedef SwarmData;
typedef SwarmMsg;

// Creates a Swaming task.
ACIPL_TASK acipl_swarm_Create();

// Cease work on this problem.  All associated
state goes
// away, and the task may no longer be used.
void acipl_swarm_End(ACIPL_TASK task);

// Pause work on the indicated task.
void acipl_swarm_Pause(ACI PL_TASK task);

// Resume work on the indicated task.
void acipl_swarm_Resume(ACIPL_TASK task);

// Creates an individual agent within the swarm.
SwarmAgent acipl_swarm_AddAgent(ACIPL_TASK task,
                        SwarmBehavior behavior);

// Connects two agents.  Only c onnected agents may
// communicate directly.
void acipl_swarm_AddNeighbor(ACIPL_TASK task,
                       SwarmAgent agent,
                       SwarmAgent neighbor);

// Disconnects two agents.
void acipl_swarm_RemoveNeighbor(ACIPL_TASK task,
                        SwarmAgent agent,
                        SwarmAgent neighbor);

// Starts executing the swarm.
void acipl_swarm_Start(ACIPL_TASK task);

// Inject data into the swarm.  This can be used
for
// initial conditions or updating information
// asynchronously.  (Alternately, the
SwarmBehavior
// can be designed to pull in data synchronously.)
void acipl_swarm_SendMsg(SwarmAgent target,
                     SwarmMsg msg);
```

Figure 64. ACIPL interface for Swarming

Preference memory looks something like this: “STM object: annotation,” where annotation is acceptable, better, best, worst, etc. Essentially, there is a fixed procedure in Soar that sorts these and determines if there is a unique candidate (evaluate_conflict_set). evaluate_conflict_set may be better as a special case of a conflict_resolution kernel.

```
// 1. Production memory
typedef struct production_struct {
  Symbol *name;
  struct production_struct *next, *prev;  // list of productions
  struct rete_node_struct *p_node;  // pointer to LHS conditions
  action *action_list;              // RHS actions
  list *rhs_unbound_variables;      // RHS vars not bound on LHS
  struct instantiation_struct *instantiations;
                                    // dll of inst's in MS
} production;

typedef ProductionMemory;

// When all the conditions of a production are satisfied, a
// dynamic data structure, the instantiation, is created to
// capture the specific variable bindings in the LHS conditions
// to elements in STM:
typedef struct instantiation_struct {
  struct production_struct *prod; // pointer to production
  struct instantiation_struct *next, *prev;
                                  // dll of inst's from same prod
  preference *preferences_generated; // header for dll of prefs
  Bool in_ms;  // TRUE iff this inst. is still in the match set
} instantiation;

// 2. Short-term memory
typedef struct wme_struct {
  Symbol *id;
  Symbol *attr;
  Symbol *value;
  Bool acceptable;
  unsigned long timetag;
  struct preference_struct *preference;     /* pref. supporting it, or
NIL */
} wme;
typedef STM;


// 3. Preference memory
enum PrefTypes {ACCEPTABLE=1,BETTER=2,BEST=3,...)
typedef struct preference_struct {
  byte type;         // acceptable, better, etc.
  Symbol *id;
  Symbol *attr;
  Symbol *value;
  Symbol *referent;
  struct slot_struct *slot;

  // dll of pref's of same type in same slot
  struct preference_struct *next, *prev;

  // dll of all pref's in same slot
  struct preference_struct *all_of_slot_next, *all_of_slot_prev;
```

Figure 65a. ACIPL outline for Soar

```
// pointer to prod instantiation
struct instantiation_struct *inst;
} preference;

do forever {

  /* Process input */

  // Add elements to the STM as determined by external situation
  // (eg, "red light")
  add_input_elements_to_stm(STM, input)

  // Remove elements to the STM as determined by external
  // situation (eg, "yellow light" no longer present)
  remove_input_elements_from_stm(STM, input)

  // In current Soar, "match" is implemented via Rete, but we
  // likely should not make this assumption for PCAA.
  // New assertions is a list of newly matching productions
  // (along with associated variable bindings).  Retractions is
  // a list of previously matching productions
  match(STM, prod_mem, new_assertions, retractions)

  /* Match/fire Productions */
  do until {no new_assertions or retractions} {
   // make assertions to STM via the actions of productions
   // can effect both content of STM and pref_mem
   fire(new_assertions, STM, pref_mem)

   // remove the assertions of previously matching prods
   retract(retractions, STM, pref_mem))

   // Do the stm changes result in new assertions or retractions?
   match(STM, prod_mem, new_assertions, retractions)
  }

  /* Choose Operator */

  // Other than match, all the algorithmic subtlety in Soar is
  // here, but this is a pretty simple procedure, essentially a
  // set of filters.
  //   1.  Remove any candidates that are not acceptable
  //       of the remainder, if there is more than one candidate
  //       remaining
  //   2.  Remove any candidates that are "worst"
  //       of the remainder, if there is more than one candidate
  //       remaining
  //   3.  Remove any candidates that are "worst than" another
  //       candidate
  //   ....

  // Number of candidates for any decision is usually small
  // (< 10).
  decision = evaluate_conflict_set(pref_mem)
```

Figure 65b. ACIPL outline for Soar

```
if {decision is operator object} {
   // Add an operator object to STM (special case of
   // add_element_to_stm)
   add_operator_to_stm(STM, decision)


   // This is really more subtle than this and looks a lot like
   // the do_until loop from above
   apply_operator(STM)
  } else {

   // Add an impasse object to STM (special case of
   // add_element_to_stm)
   add_impasse_to_stm(STM, impasse)
  }

  send_output_objects_to_external_process(STM)
}
```

Figure 65c. ACIPL outline for Soar

The main activity in Soar is to iterate around a production match-fire loop. Multiple productions can match/fire at one time. However, individual match-fire cycles result (typically) in few changes to short-term memory, relative to total size of STM, which is consistent with Rete assumptions. Input processing performs (somewhat) poorly with respect to Rete because the size of input changes are not bounded "naturally" as they are within the match/fire cycle.

7.2.1.2 General Algorithm Support

In this section we describe ACIPL support for general-purpose algorithms (i.e., those not needed specifically to support the Cog layer engines).

7.2.1.2.1 Viterbi algorithm for non-iterative solutions

Operation: Given the initial costs, and the costs of state transitions for each of "n" observation, determine the most likely sequence of states.

Calling:

```
x = acipl_viterbi(init_cost, costs, n)
```

Inputs:

- **init_costs:** Initial cost vector of starting in each state
- **costs:** Cost cube with a slice for each observation. Each slice is a matrix of transition cost from state "row" to state "col".
- **n:** The number of observations.

Outputs:

- **. x:** The determined sequence of states.

7.2.1.2.2 Viterbi for Iterative Solutions with Euclidean Distance Metric

Operation: Given the series of observations, compute the most likely state transition (and its corresponding symbol) at the depth of the decoder. Maintains state, and operates iteratively from call to call after initialization.

Calling:

```
x = acipl_viterbi_iter(obs, in_state, out_state, depth);
x = acipl_viterbi_iter(obs, 0, 0, 0); // subsequent calls
```

Inputs:
- **obs:** Observation matrix with each column having observation vector
- **in_state:** Matrix of input symbols causing state transition from row to col
- **out_state:** Cube of expected output (3rd dim) for transition from row to col. Use "inf, inf, ..." to indicate impossible transition.
- **depth:** Depth of decoder.

Outputs:
- **x:** The decoded symbols.

7.2.1.2.3 Linear Programming

Operation: Solve the linear programming problem specified in normal form. Minimizes transpose(f)*x, constraining A*x <= b.

Calling:

```
x = acipl_linprog(f, A, b);
```

Input:
- **f:** The coefficients matrix for minimization.
- **A:** The coefficients matrix for bounds constraint.
- **b:** Vector of bounds.

Output:
- **X:** The solution of the linear programming problem.

7.2.1.2.4 SAT

For SAT, an ACIPL concept API was developed, based on Reservoir's Alef SAT solver implementation. It is given below, in Figure 66.

7.2.1.2.5 Other APIs

Other ACIPL APIs discussed but not yet developed were:
- Shortest path.

- Integer programming.
- Function graph. The concept here is to implement a dataflow graph. The graph would be specified from a predetermined set of implemented functions, interconnections, and dataflow constraints (pipelineing, incremental behavior, etc).
- Heap. The usual heap operations would be supported, such as insert, remove_min, min, join, and remove.
- Hash table.
- Regression (linear and orthogonal).
- Change detection: Given two models M0 and M1 and a stream of data pick the point (if any) at which the behavior changed from M0 to M1.

Anomaly detection: Given a stream of measurements and a model, determine the point (if any) at which the measurements deviate by more than a threshold from the model.

```
typedef ACIPL_TASK;

typedef SatExpr;       // Recursive unit of statements to be
                       // satisfied

typedef SatLogicExpr;  // Expr of logic vars (kind of SatExpr)
typedef SatLogicVar;   // Logical variable (kind of SatLogicExpr)
typedef SatLogicNot;   // Logical "not" (kind of SatLogicExpr)
typedef SatLogicAnd;   // Logical "and" (kind of SatLogicExpr)
typedef SatLogicOr;    // Logical "or" (kind of SatLogicExpr)

typedef SatClause;     // A single clause to satisfy.
typedef SatClauseList; // A list of clauses.  SAT problem is
                       // one of these.

typedef SatBinding;    // A mapping of variable to value
typedef SatBindingList; // Group of bindings


ACIPL_TASK acipl_sat_Create();

SatLogicExpr acipl_sat_LogicVar();
SatLogicExpr acipl_sat_LogicNot(SatLogicExpr expr);
SatLogicExpr acipl_sat_LogicAnd(SatLogicExpr expr1,

                        SatLogicExpr expr1);
SatLogicExpr acipl_sat_LogicOr(SatLogicExpr expr1,

SatLogicExpr expr1);
//...et al.

// There are also other kinds of expressions, such as integers.
// These are provided for convenience, and are reduced internally
// to more complicated patterns of logical expressions.

// Adds a new clause to the SAT problem
SatClause acipl_sat_AddClause(ACIPL_TASK task,

SatExpr expr);

// Find only first solution (rather than all solutions)
ACIPL_TASK acipl_sat_SetFirstOnly(ACIPL_TASK task);

// Begin working on accumulated problem
void acipl_sat_Solve(ACIPL_TASK task);

// Block and wait for solution.  Returns null if no (further) solutions
```

```
      // exist.  Can be called repeatedly to get additional solutions.
      SatBindingList acipl_sat_Solution(ACIPL_TASK task);

      // Nonblocking access to partial solution.  Solutions which set
      // "is_complete" advance to the next solution
      SatBindingList acipl_sat_Solution_Partial(ACIPL_TASK task,

                                                bool& is_complete);

      // Cease work on this problem
      void acipl_sat_End(ACIPL_TASK task)
```

Figure 66. ACIPL API for SAT

7.3 Prototype Implementations

7.3.1 Code used to Synthesize POEM Memory Implementation

```
/**
 * the test amem routine
 * c - chunk data
 * v - match data
 * returns index of best match
 */
void acipl_amem(int cst[8192], int v[1024], int init)
{
  int ind;
  int cur_a, cur_b, cur_c, cur_d, cur, best;
  bool a_lt_b, b_lt_c, c_lt_d, b_lt_c_lt_d;
  int i, j, k, l, m, v0, v1, v2, v3, v4, v5, v6, v7;
  int aa0,aa1,aa2,aa3,aa4,aa5,aa6,aa7;
  int bb0,bb1,bb2,bb3,bb4,bb5,bb6,bb7;
  int cc0,cc1,cc2,cc3,cc4,cc5,cc6,cc7;
  int dd0,dd1,dd2,dd3,dd4,dd5,dd6,dd7;
  int val[512];
  int vind[512];
  int val_ab, val_cd, ind_ab, ind_cd;

  static int a0[512];
  static int a1[512];
  static int a2[512];
  static int a3[512];
  static int a4[512];
  static int a5[512];
  static int a6[512];
  static int a7[512];

  static int b0[512];
  static int b1[512];
  static int b2[512];
  static int b3[512];
  static int b4[512];
  static int b5[512];
  static int b6[512];
  static int b7[512];

  static int c0[512];
```

```
static int c1[512];
static int c2[512];
static int c3[512];
static int c4[512];
static int c5[512];
static int c6[512];
static int c7[512];

static int d0[512];
static int d1[512];
static int d2[512];
static int d3[512];
static int d4[512];
static int d5[512];
static int d6[512];
static int d7[512];

ind = 0;
best = 2147483647;

if (init) {
  for (i=0; i<256; i++) {
    j = i*32;
    k = i*32 + 8;
    l = i*32 + 16;
    m = i*32 + 24;

    a0[i] = cst[j];
    a1[i] = cst[j+1];
    a2[i] = cst[j+2];
    a3[i] = cst[j+3];
    a4[i] = cst[j+4];
    a5[i] = cst[j+5];
    a6[i] = cst[j+6];
    a7[i] = cst[j+7];

    b0[i] = cst[k];
    b1[i] = cst[k+1];
    b2[i] = cst[k+2];
    b3[i] = cst[k+3];
    b4[i] = cst[k+4];
    b5[i] = cst[k+5];
    b6[i] = cst[k+6];
    b7[i] = cst[k+7];

    c0[i] = cst[l];
    c1[i] = cst[l+1];
    c2[i] = cst[l+2];
    c3[i] = cst[l+3];
    c4[i] = cst[l+4];
    c5[i] = cst[l+5];
    c6[i] = cst[l+6];
    c7[i] = cst[l+7];

    d0[i] = cst[m];
```

```
    d1[i] = cst[m+1];
    d2[i] = cst[m+2];
    d3[i] = cst[m+3];
    d4[i] = cst[m+4];
    d5[i] = cst[m+5];
    d6[i] = cst[m+6];
    d7[i] = cst[m+7];
  }
}

else {
  v0 = v[0];
  v1 = v[1];
  v2 = v[2];
  v3 = v[3];
  v4 = v[4];
  v5 = v[5];
  v6 = v[6];
  v7 = v[7];

  for (i=0; i<512; i++)
  {
    aa0 = abs(a0[i]-v0);
    aa1 = abs(a1[i]-v1);
    aa2 = abs(a2[i]-v2);
    aa3 = abs(a3[i]-v3);
    aa4 = abs(a4[i]-v4);
    aa5 = abs(a5[i]-v5);
    aa6 = abs(a6[i]-v6);
    aa7 = abs(a7[i]-v7);
    cur_a = ((aa0+aa1)+(aa2+aa3))+((aa4+aa5)+(aa6+aa7));

    bb0 = abs(b0[i]-v0);
    bb1 = abs(b1[i]-v1);
    bb2 = abs(b2[i]-v2);
    bb3 = abs(b3[i]-v3);
    bb4 = abs(b4[i]-v4);
    bb5 = abs(b5[i]-v5);
    bb6 = abs(b6[i]-v6);
    bb7 = abs(b7[i]-v7);
    cur_b = ((bb0+bb1)+(bb2+bb3))+((bb4+bb5)+(bb6+bb7));

    cc0 = abs(c0[i]-v0);
    cc1 = abs(c1[i]-v1);
    cc2 = abs(c2[i]-v2);
    cc3 = abs(c3[i]-v3);
    cc4 = abs(c4[i]-v4);
    cc5 = abs(c5[i]-v5);
    cc6 = abs(c6[i]-v6);
    cc7 = abs(c7[i]-v7);
    cur_c = ((cc0+cc1)+(cc2+cc3))+((cc4+cc5)+(cc6+cc7));

    dd0 = abs(d0[i]-v0);
    dd1 = abs(d1[i]-v1);
    dd2 = abs(d2[i]-v2);
```

```
      dd3 = abs(d3[i]-v3);
      dd4 = abs(d4[i]-v4);
      dd5 = abs(d5[i]-v5);
      dd6 = abs(d6[i]-v6);
      dd7 = abs(d7[i]-v7);
      cur_d = ((dd0+dd1)+(dd2+dd3))+((dd4+bb5)+(dd6+dd7));

      if ( cur_a < cur_b ) {
        val_ab = cur_a;
        ind_ab = i*4;
      } else {
        val_ab = cur_b;
        ind_ab = i*4+1;
      }

      if ( cur_c < cur_d ) {
        val_cd = cur_c;
        ind_cd = i*4+2;
      } else {
        val_cd = cur_d;
        ind_cd = i*4+3;
      }

      if (val_ab < val_cd) {
        val[i] = val_ab;
        vind[i] = ind_ab;
      } else {
        val[i] = val_cd;
        vind[i] = ind_cd;
      }
    }

    for (i=0; i<256; i++)
    {
      cur = val[i];
      j = vind[i];
      if (cur < best) {
        best = cur;
        ind = j;
      }
    }

    v[0] = ind;
  }
}
```

7.3.2 POEM Training Algorithm Implementation

7.3.2.1 TSP Invariance Transformations and Initial Generation

```
% generate a random set of 1024 chunks and apply invariance transformations

n = 1024;
```

```
nt = 6;

chunks = zeros(n,nt*2);
values = zeros(n,nt);

for k = 1:n,
  x = [rand(nt,1)]; y = [rand(nt,1)];
  x=x-x(1); y=y-y(1);                          % translate to (0,0)
  r = sqrt((x-x(1)).^2 + (y-y(1)).^2);
  [max_r, max_r_ind] = max(r);
  theta = atan2(y,x);
  theta = theta - theta(max_r_ind); theta(1)=0;
  r = r ./ max(r);
  x = r.*cos(theta); y = r.*sin(theta);     % rotate and scale so last at
(1,0)

  [val,ind] = sort(r); x=x(ind); y=y(ind);  % sort by radial distance
  if y(5) < 0, y = -y; end;                 % flip if second-to-last in lower

  c = xy2cost([x y]);
  p = tsp1(c);
  chunks(k,:) = reshape([x y]',1,nt*2);
  values(k,:) = p;
end
```

7.3.2.2 POEM Replacement Policy: Largest Error

```
% This routine improves the coverage distribution of the chunks
% run on a fast machine, after running "gen_chunks"
% The routine generates some random chunks to match
% and keeps track of the one which produced the worst recalled answer.
% The new data and answer resulting in the worst match is used to replace
% the chunk in memory which has the largest accumulated mean error over.

% The following two lines are commented out so that you can run
% improve_chunks several times in succession without starting from
% the original saved chunks
%load chunks;
%load values;

n = size(chunks,1);
nt = size(chunks,2)/2;
next_max = inf;

used = zeros(n,1); % keeps track of (successes minus failures) for each row
updated = (ones(n,1));
for outer = 1:1000000 % lots of iterations -- let run overnight and ctrl-C
num_right = 0;
max_dd = 0;
for k = 1:1024,
  x = [rand(nt,1)]; y = [rand(nt,1)];
  x=x-x(1); y=y-y(1);                        % translate to (0,0)
  r = sqrt((x-x(1)).^2 + (y-y(1)).^2);
  [max_r, max_r_ind] = max(r);
  theta = atan2(y,x);
```

```
  theta = theta - theta(max_r_ind); theta(1)=0;
  r = r ./ max(r);
  x = r.*cos(theta); y = r.*sin(theta);    % rotate and scale so last at
(1,0)

  [val,ixd] = sort(r); x=x(ixd); y=y(ixd); % sort by increaseing radius
  if y(5) < 0, y = -y; end;                % flip if second-to-last in lower

  xy = [x y];
  c = xy2cost(xy);
  [p,d1] = tsp1(c);

  chunk = reshape([x y]',1,nt*2);
  score = zeros(n,1);
  for kk=3:10, score = score + (abs(chunks(:,kk)-chunk(kk))); end;
  [best_score, best_ind] = min(score);

  lp = values(best_ind,:);
  right = all(lp==p);
  num_right=num_right+right;
  if (~right)
    d2 = sum(sqrt(sum(diff(xy(lp,:)).^2,2))); dd = (d2-d1)/d1;
    used(best_ind) = used(best_ind) + dd;
    if dd>max_dd max_dd=dd; isave=best_ind; csave=chunk; psave=p; end;
  end
  updated(best_ind) = updated(best_ind)+1;
end
[v,f] = max(used./updated);
if (max_dd>v) chunks(isave,:) = csave; values(isave,:)=psave; used(isave)=0;
updated(isave)=1; end;
fprintf('%d: right=%d max=%d\n',outer, num_right,max_dd);
plot(used./updated); drawnow();

end
```

7.3.3 Rete Experimentation Library

7.3.3.1 Rete Core Algorithm

```
/**
 * rete.c    Rete algorithm interface
 * Copyright Lockheed Martin ATL
 */

#include <stdio.h>
#include "rete.h"

/**
 * Calls hash_find, and value (rather than a hash_node)
 */
int hash_find_val(hash_tree *h, char *key)
{
  hash_node *n=hash_find(h,key,strlen(key));
```

```
  return(n!=NULL ? (int) n->value : 0);
}



/**
 * read in the file and insert each line in a hash table
 */
void load_hash_table(char *fn, const char *default_fn, hash_tree *t)
{
  FILE *f;
  char s[256], *c;
  int i;

  if (!fn[0]) fn = (char *) default_fn;
  if (!(f=fopen(fn,"r"))) {fprintf(stderr,"Error on open %s\n",fn); exit(0);
}
  for (i=1, c = fgets(s,255,f); c; c = fgets(s,255,f)) {
    if (strlen(s) > 1) {
      s[strlen(s)-1]=0; /* remove newline */
      hash_insert(t,s,strlen(s),i);
      printf("adding item %s (length=%d) to hash\n",s,strlen(s));
      i++;
    }
  }
  fclose(f);
}


/**
 * hash a 3-tuple
 */
wme_t str2wme(char *s, hash_tree *h1, hash_tree *h2, hash_tree *h3)
{
  wme_t w;
  char s1[256],s2[256],s3[256];
  int v1, v2, v3;
  sscanf(s,"%s %s %s",s1,s2,s3);
  w.v1 = s1[0]=='<' ? 0 : hash_find_val(h1, s1);
  w.v2 = s2[0]=='<' ? 0 : hash_find_val(h2, s2);
  w.v3 = s3[0]=='<' ? 0 : hash_find_val(h3, s3);
  return(w);
}


/**
 * Perform a join test
 * Note: relies on WMEs being equivalent to int arrays
 */
int join_test( list_node *tests, list_node *token, wme_t *w1)
{
  test_t *t;
  int i, *wa1 = (int *)w1, *wa2, arg1, arg2;

  for ( ; tests; tests=tests->next) {
```

```
    if ( (t = (test_t *)tests->value) ) {
      arg1 = wa1[t->arg1 - 1];
      for (i=0; token && i < t->cond; i++) token = token->next;
      if ( token ) {                    // check to make sure valid wme
        wa2 = (int *) token->value;     // get arg2 wme as int array
        arg2 = wa2[t->arg2 - 1];        // get indexed field of wme
        if (arg1 != arg2) return(0);    // test binding
      }
    }
  }
  return(1);
}


/**
 * Update a list of tests according to a condition variable
 * TODO: update to handle two occurrences of same var in one condition
 */
void update_tests(list_node **tests, hash_tree *vars, char *s, int c, int f)
{
  hash_node *n = 0;
  if ( (n = hash_find(vars,s,strlen(s))) ) {
    test_t *t = calloc(sizeof(test_t),1);
    t->arg1 = f;
    t->cond = c - ((n->value)&0xffff) - 1; /* relative index */
    t->arg2 = (n->value)>>16;
    list_insert_before(*tests, t);
    n->value = c + (f<<16);   // update hash node to hold most recent
  } else {
    hash_insert(vars,s,strlen(s), c + (f<<16));
  }
}


/**
 * Print contents of a token
 */
void print_token(list_node *t)
{
  int i,j;
  wme_t *w;
  for (i=0; t ; t=t->next,i++) {
    w = (wme_t *)t->value;
    if (w) {
      for (j=0;j<i;j++) printf("  ");
      printf("(%d,%d,%d)\n",w->v1, w->v2, w->v3);
    }
  }
}


/**
 * Return a null-terminated string for the key of a given hash node
 */
char *hash_key_str(hash_node *n)
```

```
{
  char *s = malloc(n->keylen+1);
  strncpy(s,n->key,n->keylen);
  s[n->keylen]=0;
  return(s);
}


/**
 * Print string contents of a token
 */
void print_token_keys(list_node *t, hash_tree *h1,hash_tree *h2,hash_tree
*h3)
{
  int i,j;
  wme_t *w;
  char *s1,*s2,*s3;
  for (i=0; t ; t=t->next,i++) {
    w = (wme_t *)t->value;
    if (w) {
      for (j=0;j<i;j++) printf("  ");
      s1=hash_key_str(hash_search(h1,w->v1));
      s2=hash_key_str(hash_search(h2,w->v2));
      s3=hash_key_str(hash_search(h3,w->v3));
      printf("(%s,%s,%s)\n",s1,s2,s3);
      free(s1); free(s2); free(s3);
    }
  }
}
```

7.3.3.2 Rete Instrumented Hash

```
/**
 * hash.c    simple hash tree interface
 * Copyright Lockheed Martin ATL
 */

#include <stdlib.h>
#include "hash.h"
#include "timing.h"

TIMING_VAR v_find,t_find=0ull,v_ins,t_ins=0ull, v_hash,t_hash=0ull;
unsigned long long int n_find=0, n_ins=0, n_hash=0;

int NEXTPOW2(int x) {int n=1; while(n<x) n=n<<1; return(n); }


/**
 * Simple and thourough hash function (default)
 * in: n       size of hash array
 * in: key     pointer to key to be hashed
 * in: keylen  length of key in chars
 * in: seed    seed to shuffle hash for different tree levels
 */
int hash_default(int n, char *key, int keylen, int seed)
```

```
{
  int i;
  START_TIME(v_hash);
  for (i=0; i<keylen; i++)
    seed = ((seed<<1)+key[i])^((seed>>1)-key[i]);
  seed &= (n-1);
  ACC_TIME(t_hash,v_hash); n_hash++;
  return(seed);
}


/**
 * Simple and fast hash function for characters at a time
 * in: n       size of hash array
 * in: key     pointer to key to be hashed
 * in: keylen  length of key in chars
 * in: seed    seed to shuffle hash for different tree levels
 */
int hash_fast_char(int n, char *key, int keylen, int seed)
{
  int i, l= seed+1 > keylen ? keylen : seed+1;
  START_TIME(v_hash);
  for (i=0; i<l; i++)
    seed = (seed^(seed<<1))+key[i];
  seed &= (n-1);
  ACC_TIME(t_hash,v_hash); n_hash++;
  return(seed);
}


/**
 * Simple and fast hash function for integers at a time
 * Warning: Keys must be pre-padded to integral sizeof(int)
 * in: n       size of hash array
 * in: key     pointer to key to be hashed
 * in: keylen  length of key in chars
 * in: seed    seed to shuffle hash for different tree levels
 */
int hash_fast_int(int n, char *key, int keylen, int seed)
{
  int i, kl=keylen/sizeof(int), l= seed+1 > kl ? kl : seed+1, *k=(int *)key;
  START_TIME(v_hash);
  for (i=0; i<l; i++)
    seed = (seed^(seed<<1))+k[i];
  seed &= (n-1);
  ACC_TIME(t_hash,v_hash); n_hash++;
  return(seed);
}


/**
 * Create new hash tree of size "len", subsequent branching "branch"
 */
hash_tree *hash_new_tree(int len, int branch, hash_func hf)
{
```

```
  hash_tree *h = (hash_tree *)calloc(1,sizeof(hash_tree));

  h->len  = (len==0) ? 16 : NEXTPOW2(len);
  h->branch = (branch==0) ? 2 : branch;
  h->children = (hash_node **)calloc(h->len,sizeof(hash_tree *));
  h->hash = hf==NULL ? hash_default : hf;
  return(h);
}


/**
 * Return node for entry in hash tree (NULL if none found)
 */
hash_node *hash_find(hash_tree *h, char *key, int keylen)
{
  int found=0, seed=0, i;
  hash_node **t = h->children;
  START_TIME(v_find);
  i = h->hash(h->len, key, keylen, seed);
  found = t[i] && keylen==t[i]->keylen && !memcmp(key,t[i]->key,keylen);
  while ( !found && t[i] && t[i]->children ) {
    t = t[i]->children;
    i = h->hash(h->branch, key, keylen, ++seed);
    found = t[i] && keylen==t[i]->keylen && !memcmp(key,t[i]->key,keylen);
  }
  ACC_TIME(t_find,v_find); n_find++;
  return(found ? t[i] : NULL);
}


/**
 * Add entry to hash tree
 */
void hash_insert(hash_tree *h, char *key, int keylen, long value)
{
  int seed=0, i;
  hash_node **t = h->children;

  START_TIME(v_ins);
  i = h->hash(h->len, key, keylen, seed);
  while ( t[i] && t[i]->children ) { // go until no collision or end
    t = t[i]->children;
    i = h->hash(h->branch, key, keylen, ++seed);
  }

  if (t[i]) {                        // if end and collision, branch
    t[i]->children = (hash_node **)calloc(h->branch,sizeof(hash_tree *));
    t=t[i]->children;
    i = h->hash(h->branch, key, keylen, ++seed);
  }

  t[i] = (hash_node *)calloc(1,sizeof(hash_node));
  t[i]->value = value;
  t[i]->key = malloc(keylen); memcpy(t[i]->key,key,keylen);
  t[i]->keylen = keylen;
```

```
  ACC_TIME(t_ins,v_ins); n_ins++;
}


/**
 * Delete entry from hash tree
 */
void hash_delete(hash_tree *h, char *key, int keylen)
{
  int found=0, seed=0;
  int i = h->hash(h->len, key, keylen, seed);
  hash_node **t = h->children, *src;
  found = t[i] && keylen==t[i]->keylen && !memcmp(key,t[i]->key,keylen);
  while ( !found && t[i] && t[i]->children ) {
    t = t[i]->children;
    i = h->hash(h->branch, key, keylen, ++seed);
    found = t[i] && keylen==t[i]->keylen && !memcmp(key,t[i]->key,keylen);
  }

  if (found) {
    int len = seed > 0 ? h->branch : h->len;
    src = t[i]; // the node to delete
    t[i]=NULL;
    if (src->children)
      for (i=0; i<len; i++)
        if (src->children[i])
          hash_rehash(src->children[i], h);
    hash_free(h,src);
  }
}


/**
 * Rehash: Insert nodes from src tree into dst tree
 */
void hash_rehash(hash_node *src, hash_tree *dst)
{
  int i;

  if (src->children)
    for (i=0; i<dst->branch; i++)
      if (src->children[i])
        hash_rehash(src->children[i],dst);
  hash_insert(dst, src->key, src->keylen, src->value);
}


/**
 * Free a node and all its descendents
 */
void hash_free(hash_tree *t, hash_node *h)
{
  int i;

  if (h->children)
```

```
    for (i=0; i<t->branch; i++)
      if (h->children[i])
        hash_free(t,h->children[i]);
  if (h->key) free(h->key);
  if (h->children) free(h->children);
  free(h);
}


/**
 * Free hash tree structure and all its descendents
 */
void hash_free_tree(hash_tree *t)
{
  int i;

  for (i=0; i<t->len; i++)
      if (t->children[i])
        hash_free(t,t->children[i]);
  free(t);
}


/**
 * Recurse through the hash tree, looking for specified value
 */
hash_node *hash_search_recurse(hash_tree *t, hash_node *h, long value)
{
  int i;
  hash_node *ret = NULL;

  if (h->children) for (i=0; i<t->branch && !ret; i++) {
    if (h->children[i]) {
      if (h->children[i]->value==value)
        ret = h->children[i];
      else
        ret = hash_search_recurse(t, h->children[i], value);
    }
  }
  return(ret);
}

/**
 * Search and return first node found containing specified value
 */
hash_node *hash_search(hash_tree *t, long value)
{
  int i;
  hash_node *ret = NULL;

  for (i=0; i<t->len && !ret; i++) {
    if (t->children[i]) {
      if (t->children[i]->value==value)
        ret = t->children[i];
      else
```

```
        ret = hash_search_recurse(t, t->children[i], value);
    }
  }
  return(ret);
}


/**
 * Print timing results
 */
void hash_print_timing()
{
  printf("hash timing results:\n");
  RPT_TIME("find: ",t_find); printf("total find calls: %llu\n",n_find);
  printf("mean find cyc: %g\n\n",(double)t_find/(double)n_find);
  RPT_TIME("insert: ", t_ins);  printf("total insert calls: %llu\n",n_ins);
  printf("mean insert cyc: %g\n\n",(double)t_ins/(double)n_ins);
  RPT_TIME("hash: ", t_hash);  printf("total hash calls: %llu\n",n_hash);
  printf("mean hash cyc: %g\n\n",(double)t_hash/(double)n_hash);

}
```

7.4 Prototype Applications and Experiments

7.4.1 Sign of the Crescent Demonstration Problem mCML Specification

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- ================================================================
   mCML_PR.xsd : XML Schema for Plan-Recognition mCML
 -->

 <xsd:annotation>
  <xsd:documentation xml:lang="en">
   CML schema
  </xsd:documentation>
 </xsd:annotation>

<!-- ================================================================
   Basic SoC data Chunk kinds
 -->

 <xsd:element name="EntityKind" type="xsd:string"/>
 <xsd:element name="Entity">
  <xsd:complexType>
   <xsd:sequence>
      <xsd:element name="EntityName" type="xsd:string"/>
    <xsd:element ref="EntityKind"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="EventDateType">
  <xsd:choice>
```

```
   <xsd:element name="RawDate" type="xsd:string"/> <!-- xsd:dateTime -->
   <xsd:element name="ChunkDate" type="ChunkIdType"/>
  </xsd:choice>
 </xsd:complexType>

 <xsd:element name="Event">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="EventAgent" type="ChunkIdType"/>
      <xsd:element name="EventAction" type="xsd:string"/>
    <xsd:element name="EventObject" type="ChunkIdType"/>
    <xsd:element name="EventLocation" type="ChunkIdType"/>
    <xsd:element name="EventStart" type="EventDateType"/>
      <xsd:element name="EventExtent">
     <xsd:complexType>
      <xsd:choice>
       <xsd:element name="EventEnd" type="EventDateType"/>
       <xsd:element name="EventDuration" type="xsd:unsignedInt"/>
      </xsd:choice>
     </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="ListElement" type="ChunkIdType"/>
 <xsd:element name="List">
  <xsd:complexType>
   <xsd:sequence>
      <xsd:element name="ListType" type="xsd:string"/>
    <xsd:element ref="ListElement" minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

<!-- ================================================================
   Other Chunk kinds
 -->

 <xsd:element name="Inference" type="xsd:anyType"/> <!-- FIX -->

 <xsd:element name="Goal" type="xsd:anyType"/> <!-- FIX -->

 <xsd:complexType name="DataClusterType">
  <xsd:sequence>
   <xsd:element name="ClusterParent" type="xsd:string"/>
   <xsd:element name="ClusterStatistics">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="ClusterCohesiveness" type="xsd:float"/>
      <xsd:element name="ClusterSalience" type="xsd:float"/>
      <xsd:element name="ClusterTotalActivation" type="xsd:float"/>
      <xsd:element name="ClusterNumItems" type="xsd:unsignedInt"/>
      <xsd:element name="ClusterNumDataAssemblies" type="xsd:unsignedInt"/>
     </xsd:sequence>
```

```
      </xsd:complexType>
   </xsd:element>
   <xsd:element name="Items">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="Item" maxOccurs="unbounded">
       <xsd:complexType>
          <xsd:sequence>
         <xsd:element name="ItemId" type="xsd:string"/> <!-- ChunkIdType? -->
         <xsd:element name="ItemType" type="xsd:string"/>
         <xsd:element name="ItemActivation" type="xsd:float"/>
         <xsd:element name="DataAssemblyId" type="xsd:unsignedInt"/>
          </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:element name="HypothesizedLinks">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="HypothesizedLink" minOccurs="0"
maxOccurs="unbounded">
         <xsd:complexType>
          <xsd:sequence>
         <xsd:element name="Item1Id" type="ChunkIdType"/>
         <xsd:element name="Item2Id" type="ChunkIdType"/>
         <xsd:element name="MaxElementSimilarity" type="xsd:float"/>
         <xsd:element name="ItemElement1" type="xsd:string"/>
         <xsd:element name="ItemElement2" type="xsd:string"/>
          </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="CompatibilityCluster" type="DataClusterType"/>

 <xsd:element name="SimilarityCluster" type="DataClusterType"/>

</xsd:schema>

<!-- ================================================================
   End of mCML_PR.xsd
 -->
```

7.4.2 UAV Mission Planning Demonstration Problem mCML Specification

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- ================================================================
   mCML_RP.xsd : XML Schema for Route Planning
```

```
 -->

 <xsd:annotation>
  <xsd:documentation xml:lang="en">
   mCML Route Planning schema
  </xsd:documentation>
 </xsd:annotation>

<!-- ============================================================
 -->

 <xsd:complexType name="LocationType">
  <xsd:sequence> <!-- all values in "grid coordinates" -->
   <xsd:element name="X" type="AttFloat"/>
   <xsd:element name="Y" type="AttFloat"/>
   <xsd:element name="Z" type="AttFloat" minOccurs="0"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="Location" type="LocationType"/>

 <!-- describes the boundary of a rectangular area -->
 <xsd:element name="Area">
  <xsd:complexType>
   <xsd:sequence> <!-- all boundary values in "grid coordinates" -->
    <xsd:element name="XMin" type="AttFloat"/>
    <xsd:element name="XMax" type="AttFloat"/>
    <xsd:element name="YMin" type="AttFloat"/>
    <xsd:element name="YMax" type="AttFloat"/>
    <xsd:element name="GridResolution" type="AttFloat"/> <!-- meters -->
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="UAV">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Type" type="xsd:string"/> <!-- points to UAVTyp -->
    <xsd:element ref="Location"/>
    <xsd:element name="Fuel" type="AttFloat"/>
    <xsd:element name="Payload">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element ref="Weapon" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="UAVKind">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
```

```
    <xsd:element name="CruiseSpeed" type="AttFloat"/> <!-- kph -->
    <xsd:element name="StallSpeed" type="AttFloat"/> <!-- kph -->
    <xsd:element name="DashSpeed" type="AttFloat"/> <!-- kph -->
    <xsd:element name="DetectionRange" type="AttFloat"/> <!-- meters -->
    <xsd:element name="PayloadWeight" type="AttFloat"/> <!-- kg -->
    <xsd:element name="PayloadSpace" type="AttFloat"/> <!-- units -->
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="Weapon">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Type" type="xsd:string"/> <!-- points to WeaponType --
>
    <xsd:element name="Quantity" type="AttUInt" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="WeaponKind">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Speed" type="AttFloat"/> <!-- kph -->
    <xsd:element name="Range" type="AttFloat"/> <!-- meters -->
    <xsd:element name="Weight" type="AttFloat"/> <!-- kg -->
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="TimeWindow">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Start" type="xsd:dateTime"/>
    <xsd:element name="Finish" type="xsd:dateTime"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="Target">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element ref="Location"/>
    <xsd:element name="Action" type="xsd:string"/>
    <xsd:element ref="TimeWindow" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- A LocationCluster is an unordered group of atomic elements
    and/or other LocationClusters.

    The atomic elements must all be things that have a Location
```

```
    element within them: Target, Threat, or Waypoint. Note that
    different kinds may be mixed in one cluster.

    LocationClusters are hierarchical, and can contain other
    LocationClusters.
A LocationCluster can appear in a RoutePlan, as a way of
    hierarchically structuring a plan. So, it can also contain a
    RoutePlan, which should cover exactly the Targets and
    ChildClusters in this LocationCluster.

    There is some apparent redundancy between the ChildCluster and
    the RoutePlan, but the key difference is that the former is
    unordered, while the latter is ordered.
  -->
 <xsd:element name="LocationCluster">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:choice maxOccurs="unbounded">
     <xsd:element name="TargetId" type="ChunkIdType"/>
     <xsd:element name="ThreatId" type="ChunkIdType"/>
     <xsd:element name="WaypointId" type="ChunkIdType"/>
     <!-- also allow generic name, since types are just
        informational anyway -->
     <xsd:element name="LocationId" type="ChunkIdType"/>
     <!-- A cluster can contain other clusters -->
     <xsd:element name="ChildCluster" type="ChunkIdType"/>
    </xsd:choice>
    <xsd:element name="Parent" type="ChunkIdType" minOccurs="0"/>
      <!-- Parent should only be a LocationCluster -->
    <xsd:element name="Centroid" type="LocationType" minOccurs="0"/>
    <xsd:element name="Risk" type="AttFloat" minOccurs="0"/> <!-- 0 <= risk
<= 1 -->
    <xsd:element name="Cost" type="AttFloat" minOccurs="0"/> <!-- 0 <= cost -
->
    <!-- A RoutePlan, if present, should contain all of the
         elements of its parent LocationCluster: targets, threats,
         waypoints, and child clusters. -->
    <xsd:element name="RoutePlanId" type="ChunkIdType" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="Threat">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Type" type="xsd:string" minOccurs="0"/> <!-- points
to ThreatType -->
    <xsd:element ref="Location"/>
    <xsd:element name="Danger" type="AttFloat" minOccurs="0"/>
    <xsd:element name="Radius" type="AttFloat" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
```

```
 <xsd:element name="ThreatKind">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Danger" type="AttFloat" minOccurs="0"/>
    <xsd:element name="Radius" type="AttFloat" minOccurs="0"/> <!-- meters --
>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- A Waypoint can contain a TargetId, which already has a Location;
    this is why no Location is needed here if TargetId is specified.
  -->
 <xsd:element name="Waypoint">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:choice>
     <xsd:element ref="Location"/>
     <xsd:element name="TargetId" type="ChunkIdType"/>
    </xsd:choice>
    <xsd:element ref="TimeWindow" minOccurs="0"/>
    <xsd:element name="Speed" type="AttFloat" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <!-- A RoutePlan is an ordered list of points to visit. These
    points can be primarily Waypoints and LocationClusters.

    Targets themselves are also currently allowed, but we are
    considering prohibiting that and requiring them to be contained
    in their own Waypoint. The reason is that Waypoints can
    contain route-specific data that we don't want to duplicate in
    Target. (Note that a Waypoint can point to a Target already.)
    The reason we haven't made the change already is that for the
    demo problem, we may not have any of that route-specific data,
    and the extra indirection is slightly more cumbersome.
  -->
 <xsd:element name="RoutePlan">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element ref="PlanPoint" maxOccurs="unbounded"/>
     <!-- 0 <= risk <= 1 -->
    <xsd:element name="RoutePlanRisk" type="AttFloat" minOccurs="0"/>
     <!-- 0 <= cost -->
    <xsd:element name="RoutePlanCost" type="AttFloat" minOccurs="0"/>
     <!-- Parent should only be a LocationCluster -->
    <xsd:element name="Parent" type="ChunkIdType" minOccurs="0"/>
    <xsd:element name="Complete" minOccurs="0">
     <xsd:complexType>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
```

```
 </xsd:element>

 <xsd:element name="PlanPoint">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="Index" type="AttUInt"/>
    <xsd:choice>
     <xsd:element name="TargetId" type="ChunkIdType"/>
     <xsd:element name="ClusterId" type="ChunkIdType"/>
     <xsd:element name="WaypointId" type="ChunkIdType"/>
    <!-- Or if we don't want to know what type of chunk is being
         pointed to here, use a generic name instead.
       The names in the choice list above like "TargetId" are
       really just informational; there is no checking.
     -->
     <xsd:element name="LocationId" type="ChunkIdType"/>
    </xsd:choice>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

</xsd:schema>

<!-- ================================================================
   End of mCML_RP.xsd
 -->
```

7.4.3 PCAA Testbed Experiments

The PCAA Testbed is a prototype implementation of the PCAA Application (APP), Cognition (COG), and Agent Virtual Machine (AVM) layers. The Testbed gives PCAA a platform for benchmarking the performance of applications that are designed to take advantage of PCAA's capabilities.

7.4.3.1 Testbed Architecture

The Testbed implements the APP, COG, and AVM layers of the PCAA architecture in order to evaluate the performance of PCAA's UAV Mission Planning benchmark application (Figure 67). The APP layer defines the functionality and objectives of the UAV Mission Planning (MP) application. The COG layer provides problem-solving services to the APP layer in order to meet the objectives of the application. COG is divided into Proto, Micro, and Macro component layers, each of which offers unique problem solving functionality. AVM, the execution layer of the Testbed, performs computation at the request of the COG layer.

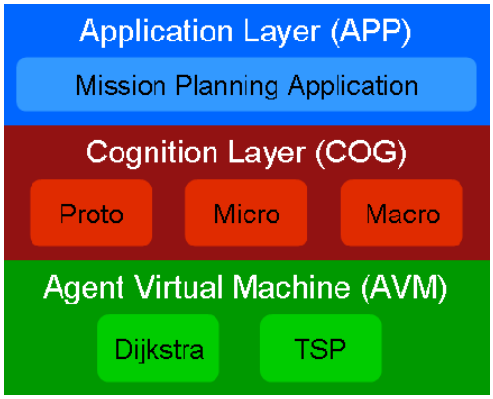


Figure 67. PCAA Testbed Architecture.

The AVM layer for the PCAA Testbed includes kernels specific to the Mission Planning application. In this implementation, we provide a Route Planning (Dijkstra algorithm) solver and a Traveling Salseman Problem (TSP) solver.

7.4.3.2 Application Layer (APP): Mission Planning Application Scenario

The target benchmark application for the PCAA Testbed is the Mission Planning application. This application simulates an Unmanned Aerial Vehicle (UAV) Mission Planning task (Figure 68). In this scenario, a UAV must plan a route through, and visit, a number of targets that are distributed across a terrain. There are threat regions within the terrain that must be avoided, as well popup threats that appear during mission execution. If a popup threat region is intersected by the UAV's flight path, the UAV must replan its route to avoid the threat, while still meeting the goal of visiting all targets.

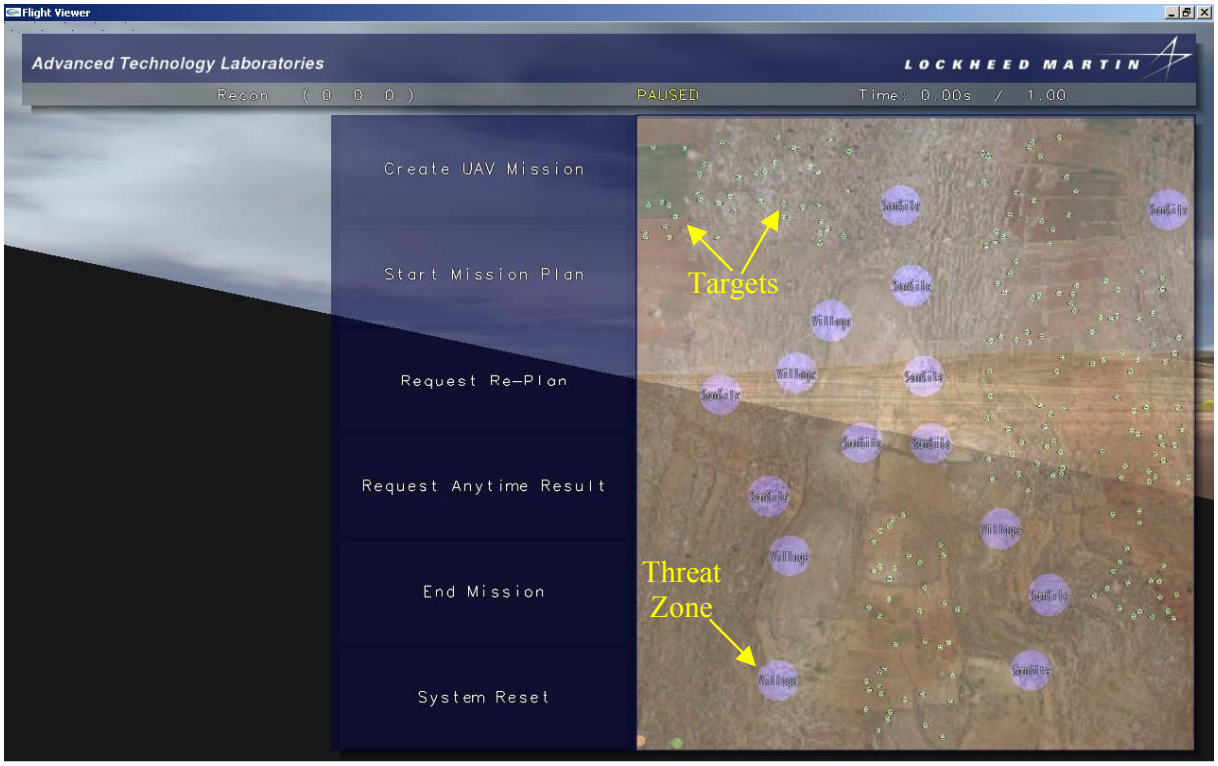


Figure 68. PCAA Mission Planning Application Flight Viewer, showing the terrain with targets and threats. Targets are green points, while threat zones are purple.

7.4.3.3 Cognition Layer (COG)

7.4.3.3.1 Proto Layer

The function of the Proto layer is to divide the task into subproblems that can be solved independently of the global problem. By reducing the global problem into smaller subproblems, Proto enables the algorithmic solvers employed in the AVM layer to operate on smaller input

sets, reducing resource requirements as well as execution time. For the Mission Planning application, Proto performs hierarchical agglomerative clustering on the scenario target list to generate meaningful subproblems (Figure 69).

Figure 69. Proto partitions target set into hierarchical subproblems, indicated here by circles.

7.4.3.3.2 Micro Layer

The Micro layer provides memoization capabilities to PCAA. Micro is responsible for learning the solutions to common subproblems that are encountered repeatedly throughout mission execution, and obtaining those solutions directly from memory rather than by recomputing them. This effectively reduces the number of invocations of our AVM kernels that are required to produce a complete solution to the global problem.

7.4.3.3.3 Macro Layer

The Macro layer generates the solution to the global problem by ordering the subproblems given by Proto, and by cooperating with Micro to request solutions to unique subproblems from the AVM kernels. Macro combines the solutions of subproblems into a solution to the entire Mission Planning task.

For each subproblem offered by Proto, Macro chooses the granularity that AVM will use when solving each subproblem. The ability to take advantage of multiple granularities in problem solving allows PCAA to generate fine-grained solutions when needed, and give reduced-complexity coarse-grained solutions when possible to save computing time and resources.

7.4.3.4 Agent Virtual Machine (AVM) Layer

Agent Virtual Machine provides kernel services to the COG layer for exact solutions to subproblems generated by the COG layer. For the Mission Planning application, AVM provides a Route Planner (Dijkstra solver) kernel, a Traveling Salesman Problem (TSP) solver, and memory for the COG layer's memoization capability (Figure 70). Route Planner computes the costs for traveling between target points. TSP computes an ordering on a set of points to minimize the cost of traveling between those points. AVM provides three separate TSP kernels. There are two heuristic solvers, both which provide a near-optimal solution to the TSP, and one optimal solver that provides an exact solution to the TSP. Due to the fact that the optimal solver can only provide solutions for small sets of points, AVM can choose the kernel to use based on tradeoffs between performance and accuracy.

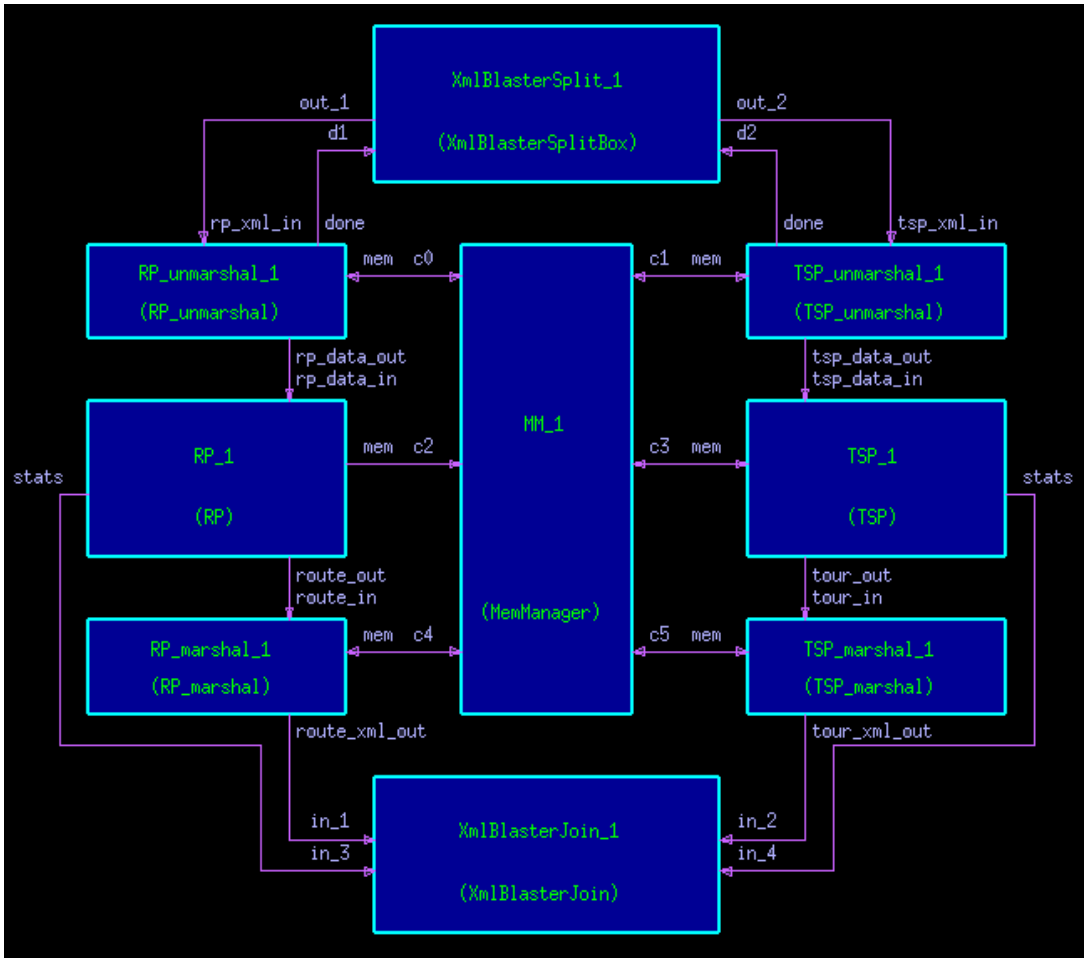


Figure 70. Agent Virtual Machine system architecture.

AVM communicates with COG via XMLBlaster XML-based message passing middleware. Requests for services arrive on the receive interface from COG, following the appropriate path based on whether it is a request for the Route Planner or TSP component. AVM then returns the results to the COG layer.

7.4.3.5 Mission Planning Benchmark

The Mission Planning benchmark requires the system to produce a mission plan through 1000 target points while avoiding threat zones. In order for the application to generate a near-optimal route through a number of points, it must generate the costs for traveling between each of the points, and then find an ordering on those points that minimizes global tour cost. Route costs are generated by the Route Planner, which uses Dijkstra's algorithm to build a table of costs between points. Then, a Traveling Salesman Problem (TSP) solver generates the ordering on those points.

The baseline benchmark implementation uses a Dijkstra solver with a fixed terrain granularity, and a heuristic TSP solver. The global mission planning problem is thus solved in a single step, rather than by solving many small problems as in PCAA.

PCAA takes advantage of the COG layer's ability to split a large problem into a number of subproblems, solve each subproblem, and then build a global solution using the solutions to the individual subproblems. A solution to the benchmark as generated by PCAA is shown in Figure 71.

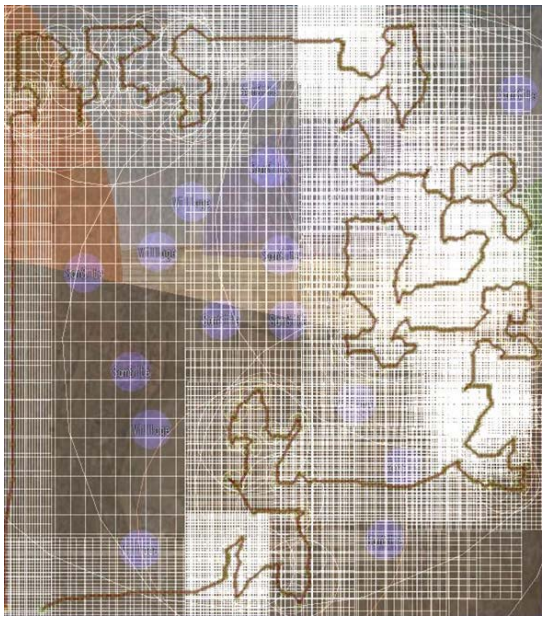


Figure 71. PCAA solution to Mission Planning benchmark application. The proposed tour is indicated by the brown line.

7.4.3.6 Results

7.4.3.6.1 Baseline Results

7.3.6.1.1 Route Planner (Dijkstra)

The baseline Dijkstra's algorithm solver results for performance and memory requirements are presented in Figures 72 and 73. Execution time and memory both increase quadratically with an increase in terrain resolution, as well as number of targets in the scenario. For 1000 targets and a terrain grid size of 150x150, we can expect execution times over 700 seconds, and memory requirements of nearly three billion bytes.

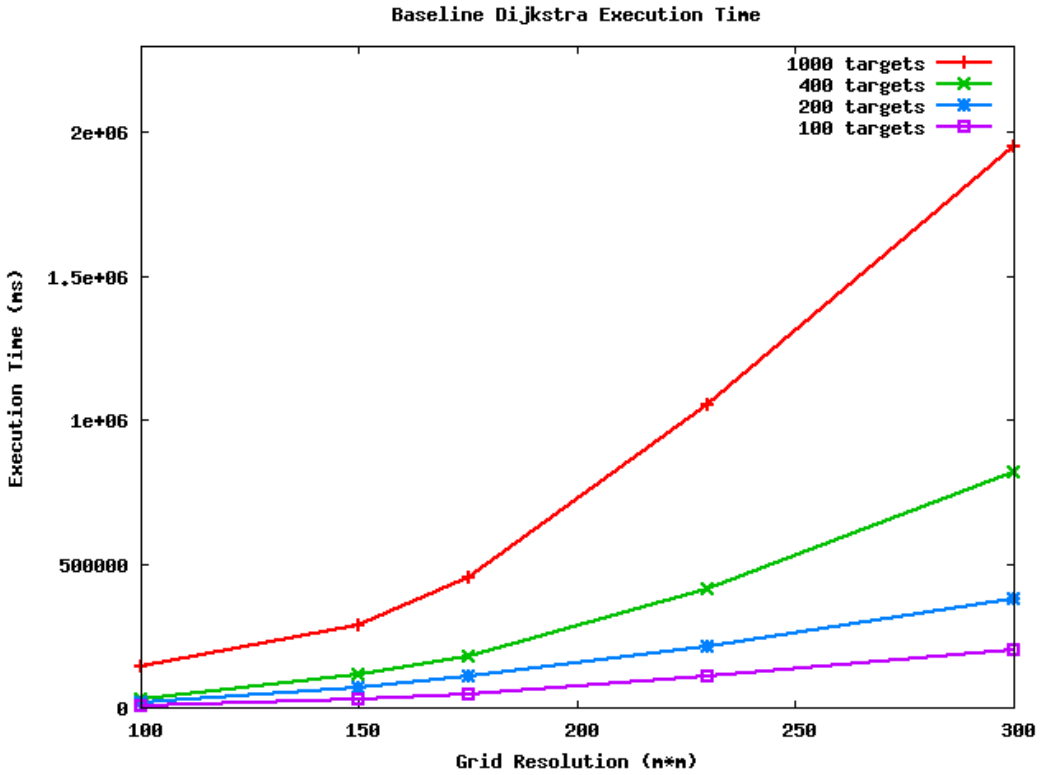


Figure 72. Execution time for the baseline implementation

Route Planner (Dijkstra) algorithm.

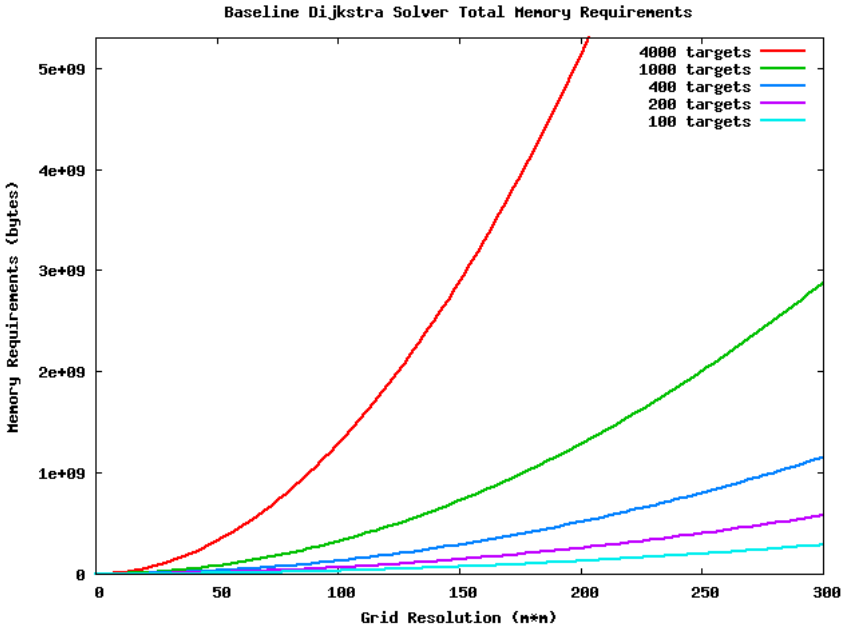


Figure 73. Memory requirements for the baseline implementation Route Planner (Dijkstra) algorithm.

7.4.3.6.2 TSP

Baseline implementation heuristic TSP execution times are presented in Figure 74. There is a quadratic increase in execution time with number of targets, and does not depend on the grid resolution used to generate costs for the tour. For 1000 points, heuristic TSP returns a result in less than one second.

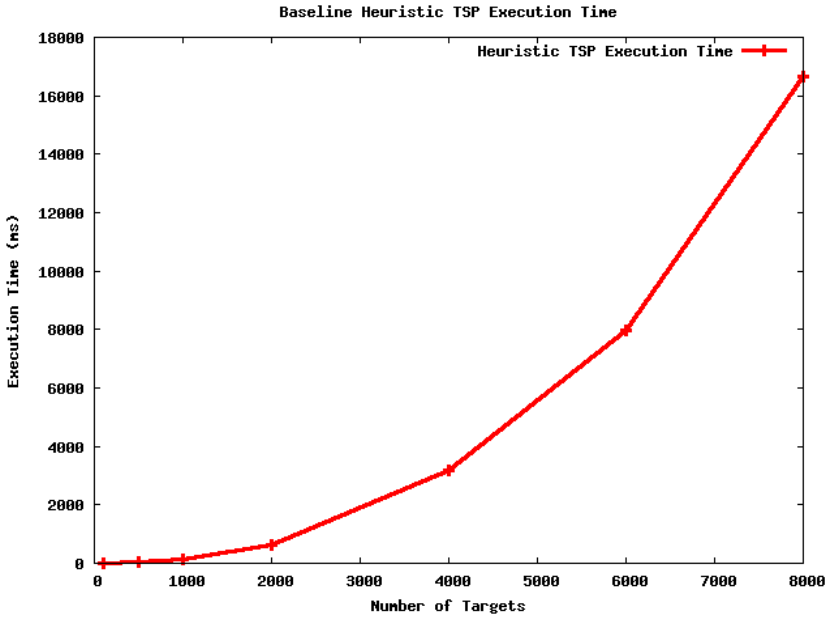


Figure 74. Execution time for the baseline implementation of the heuristic TSP.

7.4.3.7 PCAA Application Kernel Results

7.4.3.7.1 Route Planner

Performance results for the PCAA Route Planner are given in Figure 75. Due to the fact that PCAA can take advantage of multiple grid resolutions, PCAA can achieve effective grid resolutions much higher than the finest grid resolution possible with baseline. Thus, we compare PCAA's results with those of baseline with a grid resolution of 300x300. For 1000 target points, PCAA computes the route costs in about 80 seconds, while the baseline implementation requires nearly 2000 seconds.

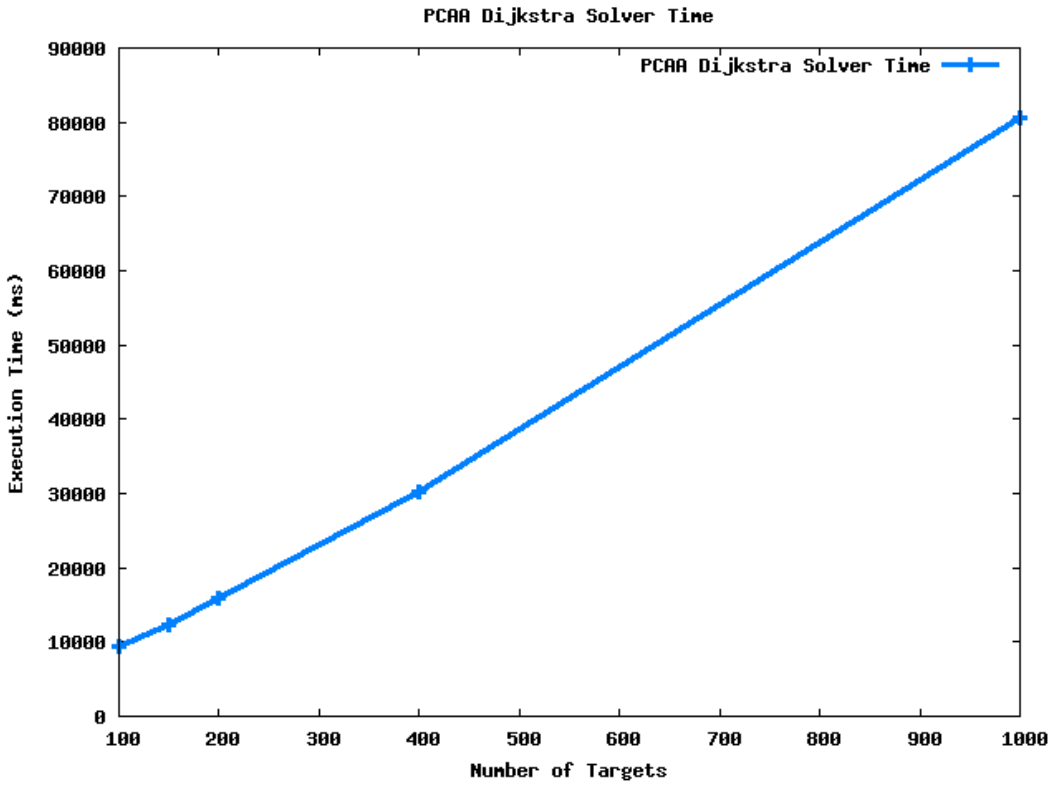


Figure 75. PCAA Route Planner (Dijkstra) solver execution time.

7.4.3.7.2 TSP

PCAA TSP solver execution times are given in Figure 76. For 1000 target points, performance of PCAA TSP is less than 600 ms, which is comparable to the baseline result.

7.4.3.7.3 Analysis

Due to PCAA's ability to split problems into meaningful subproblems, and then construct a solution of the whole out of a set of solutions to the subproblems, we can achieve linear scalability results from our Route Planner and TSP algorithms, which demand quadratic increases in execution time and memory for the baseline case.

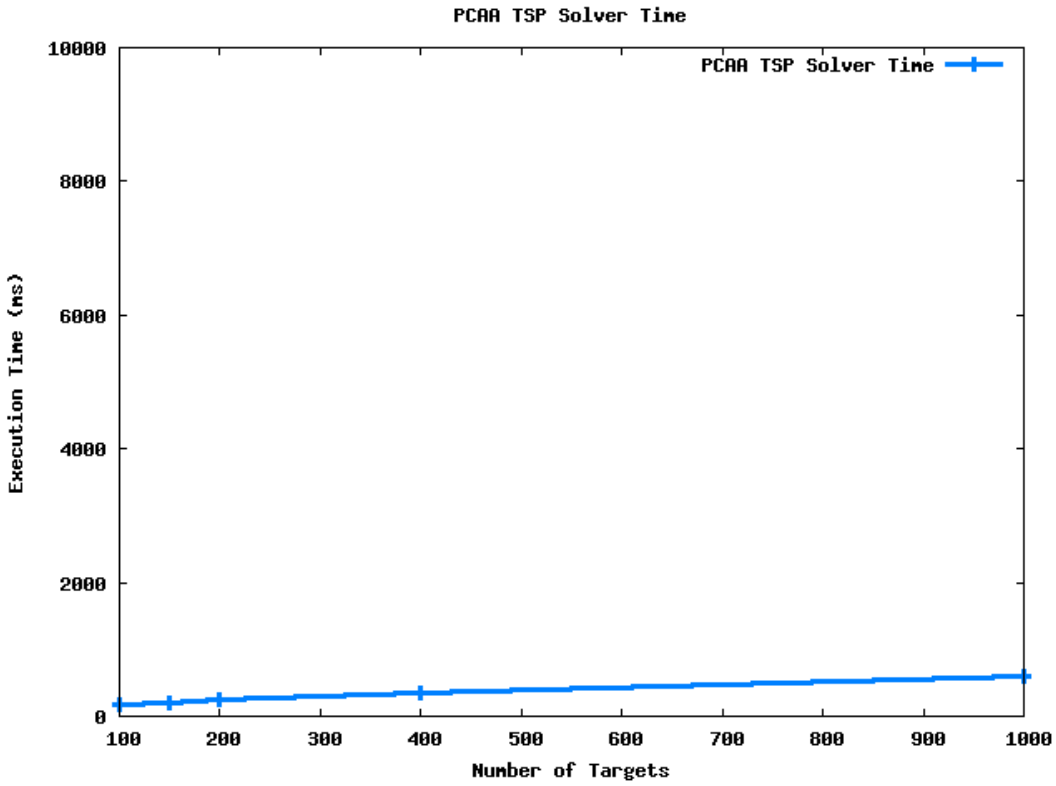


Figure 76. PCAA TSP solver execution time.

By letting PCAA choose the appropriate grid size for each subproblem rather than choosing the finest-grained grid resolution for the whole problem, we avoid the scalability issues that can prevent the baseline implementation from producing a result.

7.5 Reconfigurable Architecture for Perception

7.5.1 Executive Summary

Architectural research is increasingly driven by application needs in different domains. Research and development efforts must systematically identify the prominent properties, especially the amount of inherent parallelism and memory access/structural patterns for a wide range of emerging applications. One such application domain is recognition. Machine vision, object/scene understanding, and natural language processing among a multitude of others belong to this domain.

Recognition systems solve what is called the inverse problem. Whether it is recognizing a target from synthetic aperture radar (SAR) or image data, recognizing speech, understanding human language, identifying a person, or tracking a computer attack, the problem centers around recognizing an event, action, or object of interest in the presence of uncertainty.

Highly data-parallel preprocessing, along with multi-granularity of computation tasks, model synthesis or learning and inference, are the hallmark characteristics of machine recognition systems.

In this work we investigated a two-pronged approach to study the effect of recognition dominant of applications on computer architectures. The two-pronged approach led us to classify the inherent architecture of recognition focused applications into low and high level abstract architectures.

The low level recognition architecture supports the structural functions of recognition systems efficiently and is mainly concerned in the computation and communication aspects of recognition. The high level architecture caters to the need or constraints of specific application of recognition and is mostly concerned with meta level functions like learning, matching and searching. The main tasks or algorithms at this level are known by such names as classification, approximate matching and probabilistic computation.

In our case studies we identified efficient architectures for the two layers identified above. We performed experiments in the form of algorithm, dataset and application implementations in simulated and actual architectures. We compared these architectures to current state of the art architectures and found the graph based architecture—in particular GraphStep—to be an efficient architecture for the low level recognition layer and the active-associative memory based architecture, in conjunction with many-core architectures, to be the optimal architecture for high level or application level recognition systems.

7.5.1.1 Low Level Recognition (Structure of Recognition at Hardware Level)

Graph-oriented architectures can deliver orders of magnitude higher performance on important graph processing algorithms than conventional, monolithic processor organizations and processor based systems. These Graph Machine (GM) architectures exploit high, parallel memory bandwidth to large numbers of small, fast memories and minimize node-to-node communication time; they can sustain their high performance per leaf-processing component when assembled into large collections of components.

As silicon capacities continue to grow, these architectures are an increasingly sensible way to turn silicon capacity into increased application performance, allowing us to economically solve modest problems quickly, and making larger and larger graph problems tractable. These GMs can deliver new capabilities, with modest machines enabling real-time solution of hard problems possible in deployed, operational scenarios, and large-scale machines supporting timely management and discovery of global-scale information.

7.5.1.2 High Level (Application or Meta Level) Recognition

In this second case study, we mapped an object recognition algorithm hybrid that combines Bayesian inference with the popular geometric hashing method [Xilinx05] to our proposed multi-core active memory architecture. Geometric hashing is well-known in the computer-vision and medical imaging-research areas for its low-complexity and accuracy. The addition of heuristic Bayesian techniques, similar to those described in [Wolfson97], makes the algorithm more tolerant of fuzzy objects such as distorted images or rotational and translational variants of the same image.

The main idea in this algorithm is to compute a hash function of salient features of the object that maps to a location in an artificial "hash space" in which simple distance metrics can be computed between the hashed location and nearby entries containing library features to match against. As a result, our architecture must implement a form of approximate hashing that returns multiple entries within a cutoff distance of the hash location, and each entry is weighted by its distance. Models in our library whose features receive the highest sum of weighted hits are considered matches.

7.5.2 FPGA-Based Graph machine for Low Level Recognition

Key algorithms in cognitive computations, knowledge bases, operations research, numerical computation, computed-aided design optimization, simulation, and data-base applications traverse and manipulate sparse graph data structures. Conventional microprocessor-based systems, including clustered multiprocessors, deliver poor performance on these large, irregular graph structures; the graphs exceed on-chip cache capacities, bottlenecking performance on main-memory bandwidth and round-trip latency. To avoid this "memory wall," we developed an architecture that places graph nodes and edges in small distributed memories paired with specialized graph processing engines interconnected by a lightweight network. This architecture brings modern VLSI capabilities, including the high bandwidth and low latency available from a large number of embedded, on chip memories, to bear on sparse graph applications. Our focus study on an FPGA-based Graph Machines demonstrate that this approach yields one to two orders of magnitude speedup per leaf processing FPGA compared to a state-of-the-art Pentium processor, as well as, more robust multiprocessor scalability.

In this work, we mapped four applications to a Virtex-4 (90nm) based Graph Machine:
- Spreading activation in ConceptNet [Liu04] (speedup per leaf FPGA over 100)
- Bellman-Ford [Bellman58] shortest path search (speedup per leaf FPGA 50–100)
- Sparse Matrix-Vector Multiplication (speedup per leaf FPGA around 10)
- Conjugate Gradient (single processor speedup around 4; speedup compared to parallel Cray XD1 and IBM BG/L around 10–100)

7.5.2.1 Introduction

Irregular, sparse-graph data structures are the backbone of many efficient algorithms. In numerical computations, especially when representing physical structures (*e.g.,* circuits, finite-element analysis), sparse matrices efficiently capture the dependencies and structure in the problem. In Computer-Aided Design (CAD), the computational graph or circuit is a sparse graph, as are many auxiliary graphs derived for circuit optimization. In operations research, graphs capture resource needs, capabilities, and availability. In Artificial Intelligence, knowledge bases are efficiently represented as sparse graphs where reasoning and learning are supported by operations over sparse graphical data structures. In social networking and intelligence applications, graphs represent interactions of individuals and particular activities may be identifiable as sub-graphs possessing distinct characteristics (*e.g.,* calling or purchasing patterns that characterize weapons purchases for gangs or terrorists).

Despite their ubiquity, modern, high-performance computer architectures deliver poor performance on sparse graph applications. For example, on Sparse Matrix-Vector Multiplication (SMVM), processor-based machines typically achieve only 1–15% of their potential performance [deLorimier05]. While caching, banking, DMA block transfer, and strided prefetch allow these machines to efficiently process dense matrix operations or regular graphs, large data structures coupled with irregular data access defeat these simple optimizations. Consequently, these important applications often end up bottlenecked on main-memory bandwidth, main-memory round trip fetch latency, and high latency inter processor communications. Given the importance of sparse-graph algorithms and the poor support provided by conventional high-performance computer architectures, we ask what organization would better support these sparse-graph applications. We observe that modern silicon capacities can provide much higher bandwidth and lower latency to memory using small, distributed, on-chip memories. This organization allows us to keep all graph data local to active processing, making it efficient to visit every element in the graph or follow data-dependent paths through the graph.

7.5.2.2 Methods, Assumptions, Procedures

Our basic approach is to:

- Develop a series of representative applications (described with results and highlights in Section 3).
- For each application, identify or develop a high-performance implementation for conventional processors.
- Where possible, take best results from the literature for comparison (SMVM, Conjugate Gradient).
- Where necessary, perform our own benchmarks of conventional processors (Concept- Net, Bellman-Ford).
- For each application, develop an FPGA-based Graph Machine application including: – VHDL for FPGA application-specialized Graph Engine; use to validate area (FPGA capacity) and timing (FPGA cycle time).
- Provide an FPGA overlay network for interconnecting graph engines [Kapre06].
- Apply mapping tools to support decomposition, placement, and routing of application graph onto parallel graph machine [Kapre06] [deLorimier05].
- Provide an application router and scheduler that reports total cycles required per GraphStep operation [Kapre06] [deLorimier05].

Note that our preliminary work in [deLorimier05] [deLorimier06] was based on a Virtex-2 FPGA (150nm); the results which follow have been updated to a Virtex-4 FPGA (90nm). Further, note the earlier Sparse Matrix-Vector Multiply work used only a minimal ring network, while the more mature study reported here exploits richer butterfly fat tree networks [Kapre06] when appropriate.

7.5.2.3 Results and Discussion

Table 4 rounds up the performance benefits of our FPGA-based Graph Machine versus processor based systems for four benchmarks.

Table 4. Comparison of Execution Times on Sample Applications

App.	Dataset	Ref.	Arch.	Tech. (nm)	Clock	Peak Gflops	Leaf Units	Total Time	Total Speed Up	Per Leaf Unit	
										Speed Up	Gflops
SMVM	Matrix Market fidapm37	[Intel05] [Xilinx05]	P4-550 V4-LX160	90 90	3.4 GHz 285 MHZ	3.4 9.1	1 1 32	3.1 ms 184 µs 15.8 µs	1 17 197	1 17 6	0.50 8.33 3.04
CG	NAS CG Class A	[Intel05] * ** [Xilinx05]	P4-550 XD1 BG/L V4-LX160	90 130 90 90	3.4 GHz 2.2 GHz 700MHz 285 MHz	3.4 4.4 5.6 9.1	1 2 32 128 32	115 ms 115 ms 9ms 23ms 826 ms	1 1 13 5 140	1 0.5 0.4 0.04 4.3	0.5 0.25 0.20 0.02 2.20
Concept-Net	NYT Article	[Intel05] [Xilinx05]	P4-550 V4-LX160	90 90	3.4 GHz 285MHz	3.4	1 8 16 32	190 ms 56 µs 39 µs 27 µs	1 3412 4912 6956	1 426 307 217	N/A
Bellman-Ford	ISPD98 Ibm18	[Intel05] [Xilinx05]	P4-550 V4-LX160	90 90	3.4 GHz 285MHz	3.4	1 8 16 32	500 ms 730 406 265	1 684 1232 1888	1 86 77 59	N/A

Concept Net is a knowledge base for common-sense reasoning compiled from a Web-based, collaborative effort to collect commonsense knowledge [Liu04]. A key operation on the Concept Net knowledge base is spreading activation, also known as random walk, which computes the relatedness of concepts by calculating the probability that a random walk from a set of designated starting points will arrive at any of the other nodes in the graph. Working with the default Concept Net knowledge base, the graph has 220K nodes and 550K edges. An efficient C-implementation requires > 30 MB to represent the graph, guaranteeing it will not fit in current on-chip caches. On a typical query, a 3.4G Hz Pentium-4 implementation requires about 700 cycles (200ns) per edge, including, on average, one main memory cache miss. In contrast, the Virtex-4 implementation requires roughly three 3.5ns cycles (one receive, one update, and one send) for a total of 10.5ns per edge; we can place 64 PEs on the XC4VLX160, so each leaf FPGA is able to process 64 edges in parallel every 10.5ns. The FPGA implementation scales well to, at least, tens of leaf processing FPGAs. See [deLorimier06] for further details on the Concept Net implementation.

Bellman-Ford [Bellman58] is a single-source shortest path algorithm which robustly handles negative edge weights. The basic computation is a relaxation where each node updates its delay on each epoch to be the minimum of the distance through each of its neighbors. For a graph with n nodes, the relaxation will converge to a minimum in less than n cycles as long as there are no negative weight paths; if updates continue after n cycles, a negative weight path exists. The Graph Machine can directly implement this relaxation, with each node sending its delay to each of its neighbors during the send phase. The Graph Machine performs a global reduce operation at the end of each update phase to allow the algorithm to terminate in less than n cycles if the node values have all converged. We can fit 64 Bellman-Ford PEs on each XC4VLX160.

Iterative SMVM computes $A^i b$ by performing SMVM repeatedly with a square matrix (A). That is, it computes the recurrence:

$$b_i = A b_{i-1}$$

SMVM is the dominant computational kernel in many important iterative numerical routines including Conjugate Gradient (CG) and GMRES that solve linear constraint equations Ax = b and Arnoldi and Lanczos which solve $Ax = \lambda x$. Note that the dot products that produce each element of b_i from b_{i-1} are all independent, meaning we can parallelize across the dot products.

In our simple implementation, we partition the work by dot products. Each dot product is a graph node that we assign to memory bank/processing engine (PE) pairs (Figure 77). Conventional processors are notoriously poor at SMVM. A single processor will only yield a small fraction of its peak floating-point performance, and multiple processor ensembles are even less efficient. For example, the Pentium-4 has a peak floating-point performance of 3.4 Gflops, but only achieves 0.5 Gflops on this task. The iterative SMVM is an example where the entire graph (the entire matrix) must be processed in each iteration. Consequently, if the matrix is too large to fit into fast, on-chip caches, performance is bound by the higher latency and lower bandwidth of off-chip memory access. Parallel versions could, potentially, reduce the per processor working set; however, communication often ends up dominating computation due to high end-to-end network latency and high network contention.

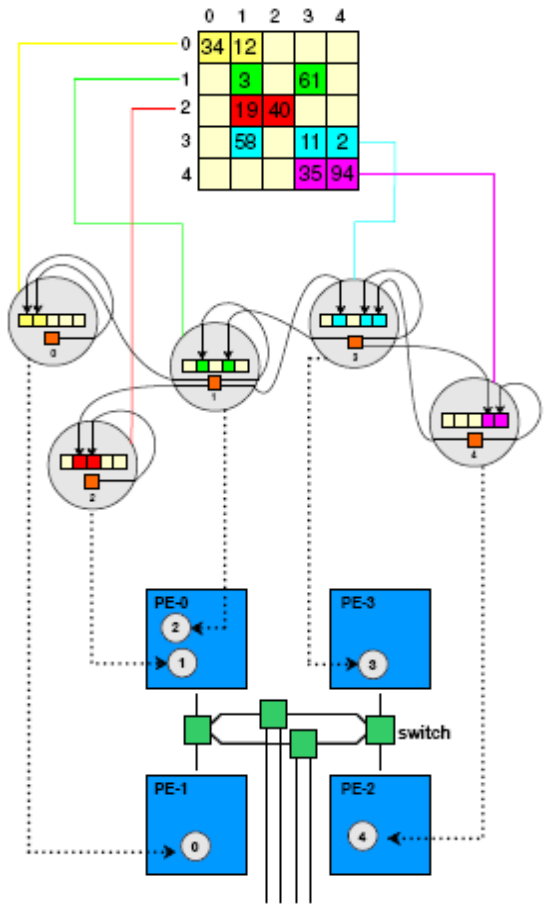


Figure 77. Assign Matrix Row Dot-Products (Graph Nodes) to PEs for Parallelism

Our FPGA-based Graph Machine implementation is able to perform better because of the high memory bandwidth [deLorimier06] and low PE-to-PE latency. On a Virtex4-LX160-12, we are able to place16 double-precision floating-point PEs which operate at 285MHz each. This gives a raw floatingpoint performance per chip of 9 double-precision Gflops which is higher than the Pentium-4 peak floating-point performance at the same technology node (90nm). We store all data adjacent to the PEs in the embedded memory blocks so we can feed the PEs at their peak operating rate. A single Virtex4-LX160 can sustain 8.3 Gflops on SMVM. For multiple-chip implementations, we use a low-latency time-multiplexed, butterfly fat-tree network to route among PEs. Using 32 leaf processing FPGAs (512 PEs), we are able to sustain a per leaf processing rate of 3 Gflops. More details on our first-generation FPGA-based SMVM implementation are reported in [deLorimier05].

Conjugate Gradient (CG) is an iterative algorithm for solving a set of linear equations (Ax =b) whose matrix, A, is symmetric and positive definite. Since CG is one of the NAS benchmarks, conventional processors and supercomputers are regularly rated by their CG performance. The dominant computation in each iteration of CG is an SMVM as described in the previous section; the SMVM is followed by dot products, vector addition, and scaling on vectors with length equal to the length of x.

7.5.2.4 Conclusions

Graph-oriented architectures can deliver orders of magnitude higher performance on important graph processing algorithms than conventional, monolithic processor organizations and processor based systems. These GM architectures exploit high, parallel memory bandwidth to large numbers of small, fast memories and minimize node-to-node communication time; they can sustain their high performance per leaf-processing component when assembled into large collections of components.

As silicon capacities continue to grow, these architectures are an increasingly sensible way to turn silicon capacity into increased application performance, allowing us to economically solve modest problems quickly, and making larger and larger graph problems tractable. These Graph Machines can deliver new capabilities, with modest machines enabling real-time solution of hard

problems possible in deployed, operational scenarios, and large-scale machines supporting timely management and discovery of global-scale information.

7.5.2.5 Recommendations

Graph Machine architecture demonstrates real benefits for important cognitive and scientific applications. As the work here demonstrates, with effort, a certain level of benefit is exploitable with today's commercial, off-the-shelf technologies. However, to fully exploit these capabilities, further research and development is needed, including:

- Explore/estimate impact of customization—custom graph-machine designs rather than simply overlies on existing FPGAs; we expect custom designs might offer an additional order of magnitude benefit in performance per die and energy per operation along with multiple orders of magnitude improvement in scalability.
- Further explore suitable intra-chip and inter-chip network design and optimization for this architecture and these applications.
- Develop suitable languages, programming models, system architectures, and tools to ease the development and automate the exploitation of these capabilities.
- Develop/Analyze further applications.
- Explore integration of core graph processing into larger system contexts.

7.5.3 Active Memory Based Architecture For Application Level Recognition

Ideally, a recognition system will perform library matching or classification based on features extracted from a model generated by an appropriate sensor. Regardless of the particular recognition application or the complexity of the sensor, recognition systems typically perform some type of data preprocessing, feature extraction, and matching or classification. While the first two stages can be computationally intensive, the final stage of matching or classification is memory access intensive. As the gap between memory and processor speed grows, by Little's Law the amount of concurrency needed to hide the latency of memory accesses will continue to increase. Memory access therefore quickly becomes the bottleneck in the implementations of the inference mechanism for matching or classification. To attack this problem, designers must focus on innovative implementations of algorithms and architectures, such as embedding memory alongside the inference logic. In this work, we demonstrate such an implementation for a geometric Bayesian inference algorithm applied to object recognition and a full fledged graph based architecture. Our particular mapping accounts for the memory bottleneck and adapts a memorization-based architecture to implement the inference logic. We simulate this implementation on the framework and demonstrate its effectiveness as compared to a multi-core implementation with a monolithic memory structure. Our simulation results also indicate optimal sizes for the embedded memory for various object recognition benchmarks.

In this second case study, we mapped an object recognition algorithm hybrid that combines Bayesian inference with the popular geometric hashing method [Xilinx05] to our proposed multi-core active memory architecture. Geometric hashing is well-known in the computer-vision and medical imaging-research areas for its low-complexity and accuracy. The addition of heuristic Bayesian techniques, similar to those described by Wolfson and Rigoutsos [Wolfson97], makes

the algorithm more tolerant of fuzzy objects such as distorted images or rotational and translational variants of the same image.

The main idea in this algorithm is to compute a hash function of salient features of the object, that map to a location in an artificial "hash space" in which simple distance metrics can be computed between the hashed location and nearby entries containing library features to match against. As a result, our architecture must implement a form of approximate hashing that returns multiple entries within a cutoff distance of the hash location, and each entry is weighted by its distance. Models in our library whose features receive the highest sum of weighted hits are considered matches.

7.5.3.1 Active Memory Architectural Mapping

One obvious choice for implementing geometric hashing is the Content Addressable Random Access Memory (CA-RAM) architecture [Rigoutsos95], that places a hash function logic alongside conventional memory structures. Although this architecture can, in principle, be extended to perform the approximate hybrid approach, a natural extension to CA-RAM that incorporates approximate matching is the Programmable Object Evaluation Memory (POEM) architecture [Cho07].

The POEM architecture (Figure 78) matches stored content to an input feature vector (operand) through the user-definable distance function "F." Our first implementation of POEM on a Xilinx Virtex-2 has four, dual-port banks of memory, each 256 bits wide by 512 words deep. Each 256-bit row of memory represents eight integers of stored feature content. The four banks of memory are compared to new input feature vectors in parallel, according to the specified function, and the results are fed into a pipelined minimization tree. The data object associated with the best-responding row is returned.

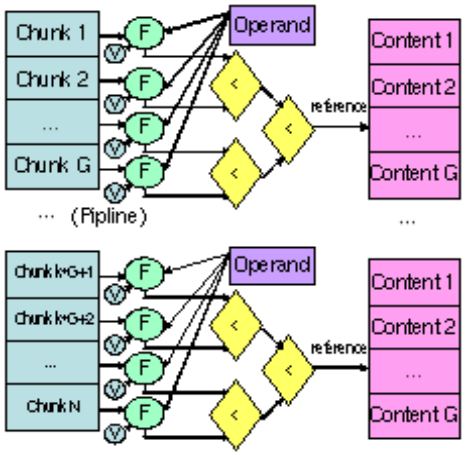


Figure 78. POEM Architecture

The Traveling Salesman Problem (TSP) was chosen to test the recall ability and accuracy of such a memory. In learning mode, candidate problems are solved optimally, and the problem/solution pairs are stored. A replacement policy, which favors replacing least frequently used content with new inadequately represented content, flattens the response error.

In this application, "F" implements the hash distance, and the subsequent POEM logic performs the weight summations.

To effectively scale a multi-core POEM architecture to the geometric inference problem, we must ensure that the match logic does not idle due to data starvation. This unwanted behavior often occurs in a multi-core architecture with a monolithic memory structure in which the bottleneck quickly becomes the communication between memory and the cores (Figure 79). One way to address this issue is to split the memory into chunks, where each chunk is dedicated to a

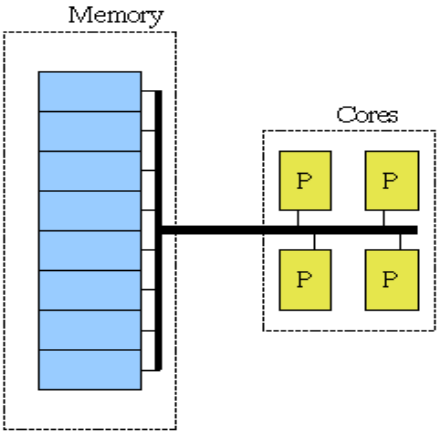


Figure 79. A multi-core architecture with a monolithic main memory.

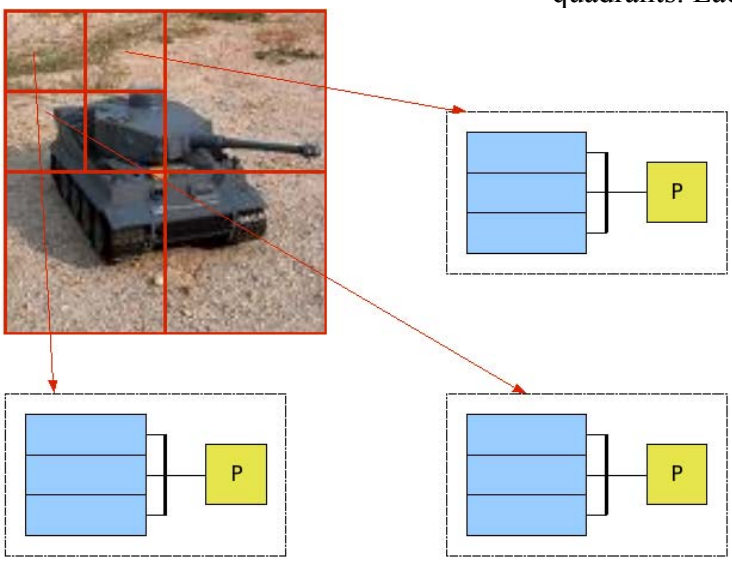


Figure 80. The proposed architecture for Bayesian inference object recognition. The image is partitioned recursively into quadrants, and each region (addressable by row and column bits) is assigned to a core and memory chunk.

particular core. Since each core and memory chunk are independent in this scheme, having a systematic way of assigning each core to a separate piece of the problem is important.

The way chunking memory is done and the effectiveness of the approach varies with the problem. In the case of Bayesian inference for object recognition, the problem can be partitioned based on feature location within the image. In our approach, we recursively divide up the relevant features of images in the library into quadrants (Figure 80). The smallest regions are addressed via row and column bits, where each bit represents recursion into a smaller set of quadrants. Each of the smallest regions are assigned to a core and memory chunk. When matching occurs, features from the image under analysis are dispatched to the appropriate core and memory chunk based on the location of the feature. This dispatching process is efficient and requires little logic since it is based on nothing more than a grid discretization of the image.

A key parameter that must be adjusted to optimize performance in our approach is the memory-chunk size. This parameter depends not only on the location density of image features, but also on the available number of cores and the desired granularity. In other words, more cores can be used to either match against larger image regions in larger images (potentially requiring larger memory chunks) or to match against smaller regions in smaller images (an increase in the granularity, which potentially requires smaller memory chunks). We explore this critical tradeoff via simulation using JHDL framework [Russo06]. Figure 81 shows the POEM chuck implementation in JHDL and Table 5 shows the preliminary results.

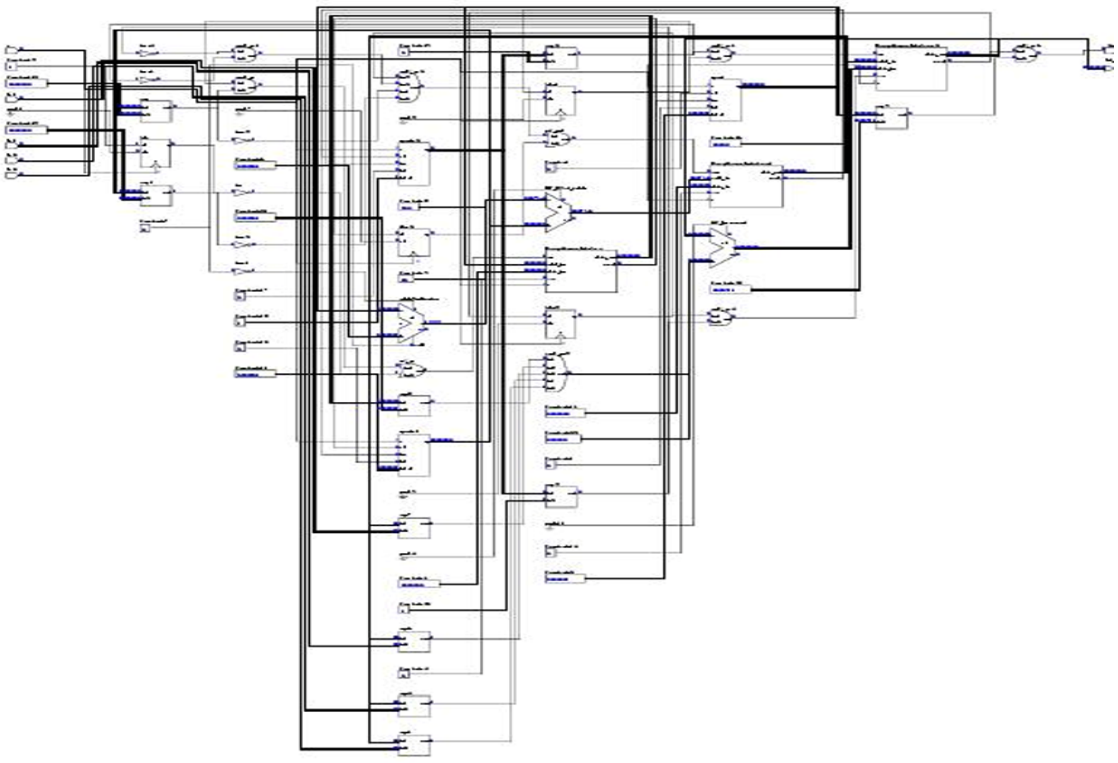


Figure 81. POEM chuck implementation in JHDL

Table 5. JHDL Preliminary Results

Memory R/W Cycle Count	JHDL Simulation Runtime (s)	1 Processor Cycle Count	2 Processor Cycle Count (Approximate)	4 Processor Cycle Count (Approximate)	8 Processor Cycle Count (Approximate)
1	29.032	297927	148964	74482	37241
2	35.953	372409	186205	93103	46552
5	56.562	595855	297928	148964	74482
10	91.016	968265	484133	242067	121034
50	366.812	3947545	1973773	986887	493444
100	708.031	7671645	3835823	1917912	958956
150	1051.672	11395745	5697873	2848937	1424469
200	1400.625	15119845	7559923	3779962	1889981
250	1740.453	18843945	9421973	4710987	2355494
300	2081.031	22568045	11284023	5642012	2821006

7.6 References

[Bellman58] Bellman, Richard. On a Routing Problem, Quarterly of Applied Mathematics 16(1), pp. 87-90, 1958.

[Cho07] Cho, S., J.R. Martin, R. Xu, M.H. Hammoud, and R. Melhem, CA-RAM: A High-Performance Memory Substrate for Search-Intensive Applications. In Proc. of the IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS 2007), pp. 230-241, 2007.

[deLorimier05] deLorimier, Michael and Andr´e DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. In Proceedings of the International Symposium on Field-Programmable Gate Arrays, pp. 75–85, February 2005.

[deLorimier06] deLorimier, Michael, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tom´as E. Uribe, Thomas F. Knight, Jr., and Andr´e DeHon. GraphStep: A System Architecture for Sparse-Graph Algorithms. In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 243–151. IEEE, 2006.

[Horita00] Horita, T. and M. Wakabayashi, Environment for Multiprocessor Simulator Development. In Proc. of the Intl. Symposium on Parallel Architectures, Algorithms and Networks (ISPAN 2000), p. 64, 2000.

[Intel05] Intel Corporation Intel Pentium 4 Processor Product Briefs. http://www.intel.com/deign/Pentium4/profbref/, December 2005.

[Kapre06] Kapre, Nachiket, Nikil Mehta, Michael deLorimier, Raphael Rubin, Henry Barnor, Michael J. Wilson, Michael Wrighton, and Andr´e DeHon. Packet-Switched vs. Time-Multiplexed FPGA Overlay Networks. In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 205–213. IEEE, 2006.

[Lauer93] Lauer, Peter, Functional Programming, Concurrency, Simulation, and Automated Reasoning. Springer Verlag, 1993.

[Liu04] Liu, Hugo and Push Singh. ConceptNet – A Practical Commonsense Reasoning Tool-Kit. BT Technical Journal, 22(4):211, October 2004.

[Milner90] Milner, Robin, Mads Tofte, Robert Harper. The Definition of Standard ML. MIT Press 1990.

[Milner97] Milner, Robin, Mads Tofte. Commentary on Standard ML. MIT Press, 1997.

[Pfeffer01] Pfeffer, A. IBAL: An Integrated Bayesian Agent Language. *Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[Ramsey02] Ramsey, N., A. Pfeffer. Stochastic Lambda Calculus and Monads of Probability, POPL 2002.

[Rigoutsos95] Rigoutsos I. and R. Hummel. A Bayesian Approach to Model Matching with Geometric Hashing. Computer Vision and Image Understanding, Vol. 62, No. 1, pp. 11-26, 1995.

[RStream] RStream high level complier, http://www.reservoir.com

[Russo06] Russo, J.C., M. Amduka, K. Pederson, R. Lethin, J. Springer, R. Manohar, and R. Melhem. Enabling Cognitive Architectures for UAV Mission Planning. In Proc. of the Tenth Annual High Performance Embedded Computing Workshop (HPEC 2006), 2006.

[Wolfson97] Wolfson, H.J. and I. Rigoutsos. Geometric Hashing: An Overview. IEEE Computational Science and Engineering, Vol 4, No. 4, pp. 10-21, 1997.

[Xilinx05] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. Xilinx Virtex-4 Family Overview, June 2005. DS112 http://direct.xilinx.com/bvdocs/publications/ds112.pdf

8. Glossary

A* solver: A solver that finds its solution(s) through A* search. A* search is a graph search algorithm that works by assigning heuristic "goodness" estimates to nodes at a level and then by searching the graph by picking the "best" node at that level.

Abduction: It is a reasoning technique to derive the underlying facts leading to the seen evidences/conclusions.

ACIP (Architecture for Cognitive Information Processing)

ACIPL (Advanced Cognitive Information Processing Library)

ACT-R (Adaptive Control of Thought – Rational): A cognitive architecture designed by John Anderson, Christian Lebiere, and many others at Carnegies Mellon University.

AFPGA (Asynchronous FPGA)

Amdahl's law: This is a formula to estimate expected speedup from parallelizing an algorithm.

Anytime solver: A solver that can produce, for a given problem, an improving-quality solution over time such that an approximate solution is available at all times.

API (Application Programmer's Interface)

APL (Asynchronous Programmable Logic)

ASIC (Application-Specific Integrated Circuit)

ATO (Air Tasking Order)

AVM (Agent Virtual Machine Layer (of PCAA))

Backgammon: It is a boardgame for two players, with pieces moving according to the throw of a dice.

Bayesian statistics: A technique to derive statistical probability of an event from the statistical probability of a related event.

C3I1 (Three Cognitions, One Integrated Architecture): the name for the PCAA Cognitive Layer architecture

C4ISR (Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance)

CAOC (Combined Air Operations Center)

CA-RAM (Content-Addressable RAM)

CHASM: the name for PCAA Hardware Layer architecture

CML (Cognitive Markup Language)

CMOS (Complementary MOS (Metal Oxide Semiconductor))

COAC (Combined Air and Space Operations Center)

Cognitive pyramid: A term to describe the information-flow through the PCAA Cognitive Layer such that, for a given problem, the simplest reasoning is applied to the largest amount of data, and the most complex reasoning is applied to the smallest amount of data.

CRCW Memory (Common Read Common Write memory)

Deduction: It is a reasoning technique to derive conclusions from the underlying facts.

DMPI (Designated Mean Point of Impact)

DRAM (Dynamic RAM)

DSM (Distributed Shared Memory)

EA (Electronic Attack)

EWO (Electronic Warfare Officer)

FLOPS (Floating Point Operations Per Second)

FPGA (Field Programmable Gate Array)

GFLOPS (Giga FLOPS)

GPS (Global Positioning System)

HAC (Hierarchical Ant Clustering algorithm)

JDAM (Joint Direct Attack Munition)

JFCOM (Joint Forces Command)

LAR (Launch-Acceptable Region)

LPI (Low Probability of Intercept)

LSA (Latent Semantic Analysis)

MAC (Multiply and Accumulate)

mCML (machine Cognitive Mockup Language)

Markovian Assumption: The assumption that the outcomes from the current state are dependent on its immediately previous state only.

MPI (Mean Point of Impact)

NCC (Network-centric Communication)

NP-complete (Nondeterministically Polynomial complete): A class of (hard) problems which, as far as anyone currently knows, can be solved in polynomial time only through nondeterministic algorithms. The "complete"-part refers to the class of problems such that solving any one problem in this class can solve all the other problems in this class.

OIL (Ontology Interface Layer)

OWL (Semantic markup language for interfacing with ontologies, based on DAML+ OIL)

PCAA (Polymorphous Cognitive Agent Architecture)

Production learning: Learning the production rules.

PE (Processing Element)

POEM (Programmable Objective Evaluation Memory)

PSCM (Problem Space Computational Model)

RAM (Random Access Memory)

RBS (Rule-based System)

RCS (Radar Cross-Section)

SAM (Surface-to-Air Missile)

SAR (Synthetic Aperture Radar; or, Search and Rescue mission)

SDA (Sense-Decide-Act cycle of Soar)

SMVM (Sparse Matrix-Vector Multiplication)
\
SoC (Sign of the Crescent problem)

SODAS (Self-Organizing Data and Search)

TOT (Time over Target)

TSP (Traveling Salesman Problem)

UAV (Unmanned Air Vehicle)

UCAV (Unmanned Combat Air Vehicle)

UMP (UAV Mission Planning problem)

URAV (Unmanned Reconnaissance Air Vehicle)

USAF (U.S. Air Force)

VHSIC (Very Highspeed Integrated Circuit)

VHDL (VHSIC Hardware Description Language)

VLIW (Very Long Instruction Word)

9. Index